AFIT/GCS/ENG/99M-09

Transforming Aggregate Object-Oriented Formal Specifications to Code

THESIS
John A Kissack
1Lt, USAF

AFIT/GCS/ENG/99M-09

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE
TRANSFORMING AGGREGATE OBJECT-ORIENTED FORMAL SPECIFICATIONS TO CODE

5. FUNDING NUMBERS

6. AUTHOR(S)
JOHN A. KISSACK, 1Lt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/99M-09

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Roy F. Stratton
AFRL/IFTD
525 Brooks Rd.
Rome, NY 13441-4505
(303) 315-3004 (DSN 587-3004)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Dr. Thomas C. Hartrum
(937) 255-3636 x4581
Thomas.Hartrum@afit.af.mil

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The feasibility of a transformational formal-based software engineering tool has been the focus of AFIT research for several years. Until now, the main research emphasis has been placed on the individual components that would comprise such a transformational system; therefore, this research demonstrates how a representative collection of aggregate objects would be transformed from specification to code. The research focused on critical integration issues associated with a formal-based software transformation system, such as the source specification, the problem space architecture, design architecture, design transforms, and target software transforms.

Software is critical in today's Air Force, yet its specification, design, and development have not achieved a satisfactory level of reliability or consistency. Techniques such as formal-based methods apply sound engineering principles to software development, greatly increasing software's quality and reliability.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software engineering, Transformation systems, object-oriented, Z specifications, Specification analysis, Design analysis	101
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

AFIT/GCS/ENG/99M-09

Transforming Aggregate Object-Oriented Formal Specifications to Code

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

John A Kissack, B.S. Computer Science

1Lt, USAF

March, 1999

AFIT/GCS/ENG/99M-09

Transforming Aggregate Object-Oriented Formal Specifications to Code

John A Kissack, B.S. Computer Science

1Lt, USAF

Approved:

Thomas C. Hartrum	4 Mar 99
Dr. Thomas C. Hartrum Committee Chair	Date
Robert P. Graham	4 MAR 99
Maj. Robert P. Graham Jr. Committee Member	Date
Henry B. Potoczny	4 MAR 99.
Dr. Henry B. Potoczny Committee Member	Date

Acknowledgements

There are a number of individuals that made my thesis effort a success. First and foremost, I would like to thank both my thesis advisor, Dr. Thomas Hartrum, whose deft guidance and infinite patience made this research possible, and committee member, Major Robert Graham, for providing his expert opinion on formal methods, yet allowing my 'Highly Unorthodox' approaches to be explored. I would like to thank PhD student Major Tom Schorsch for his sometimes cryptic, always helpful opinions. I also owe a debt of gratitude to fellow student Travis Tankersley. His valuable inputs to our frequent brainstorming sessions, combined with his tireless effort implementing primitive objects, directly contributed to the success of my research.

I would like to thank my father- and mother-in-law, Orrin and Melva Olk, for their unwavering support and guidance throughout my career.

Most importantly, I would like to thank my wife Karen, daughters Ashley and Rebecca, and sons Adam and Kyle, for their understanding and patience during these past eighteen months.

John A Kissack

Table of Contents

Page

Acknowledgements ii

List of Figures vi

Abstract viii

I. Introduction 1

1.1 Introduction 1

1.2 Problem 2

1.3 Scope 4

1.4 How This Document is Structured 4

II. Background 5

2.1 Introduction 5

2.2 Informal Object-Oriented Specification Modeling 5

2.2.1 Informal Aggregate and Association Specification Modeling 6

2.3 Formal Specification Modeling 7

2.3.1 Model-Based Aggregate and Association Specification Modeling 7

2.3.2 Formal Functional Specification Modeling 10

2.3.3 Formal Dynamic Specification Modeling 10

2.3.4 Restricting Z Specifications Using Declarations and Invariant Constraints 12

2.4 Software Design 14

2.5 Relationship Between Software Specification and Design . . . 15

2.6 Software Development Transformation 15

Page

2.6.1 Aggregate and Association Structure Transformation 16

2.7 Categorizing Transformations 16

2.8 Assessment of Previous AFIT Research 17

III. The Specification Model 20

3.1 Introduction 20

3.2 The Existing Primitive Specification Domain Model 20

3.2.1 Domain Model Structure 20

3.2.2 Functional Aspects of the Domain Model. 22

3.2.3 Dynamic Aspects of the Domain Model. 23

3.3 Domain Model Additions to Support Aggregation 23

3.3.1 Class Relationship Structure 24

3.3.2 Aggregate Class Structure 25

3.3.3 Association Structure 25

3.3.4 Event Map Structure 26

IV. The Design Model 27

4.1 Introduction 27

4.2 Analysis 27

4.2.1 The GOM 29

4.2.2 Representing Aggregation in the GOM 30

4.3 Extensions to the GOM 31

V. Design Transforms 33

5.1 Introduction 33

5.2 Transformation Assumptions 33

5.3 System Level Decisions 34

5.4 Overall Aggregate Transformation Process 36

5.5 Container Specifications 37

Page

5.5.1 Container Requirements 37

5.5.2 Set Container Requirement Definition 37

5.5.3 Sequence Container Requirement Definition 38

5.5.4 Association Container Requirement Definition . . . 38

5.5.5 Containers As Implemented 40

5.6 Aggregate Component Attribute Transformations 41

5.7 Aggregate Invariant Constraint Transformation 43

5.8 Set, Sequence, and Association Operations Transformation . 50

5.9 Aggregate Predicate and Expression Transformation 56

5.10 Summary 58

VI. Conclusions and Possible Research 59

6.1 Results 59

6.2 Limitations 60

6.3 Lessons Learned 61

6.4 Follow-on Research 62

6.5 Summary 63

Appendix A. Attribute Transformation Example - Specification 64

Appendix B. Transforms 4, 5, 6, and 7 Examples - Specification 76

Appendix C. Transforms 9 and 13 Examples - Specification 81

Bibliography 88

Vita 90

List of Figures

Figure		Page
1.	Bicycle Aggregate Example	6
2.	Single Aggregate Component Example	8
3.	Component Set Example	9
4.	Component Association Example	9
5.	Functional Schema Example	10
6.	State Transition Table Example	11
7.	Association Multiplicity	13
8.	Function Venn Diagram	14
9.	Software Transformation System	18
10.	AFITTOOL Component Layout	19
11.	Domain Structure	21
12.	Domain Inheritance Hierarchy	21
13.	Domain Derived Type Example	22
14.	Connection Structure	24
15.	Aggregate Class Structure	25
16.	Association Class Structure	25
17.	Event Mapping Structure	26
18.	Design Inheritance Hierarchy	27
19.	Design Structure	28
20.	Design Program Constructs	30
21.	Aggregate Transformations	37
22.	Mandatory Set Container Methods	38
23.	Mandatory Sequence Container Methods	38
24.	Mandatory Association Container Methods	39
25.	Association Domain and Range Composition	39

Figure		Page
26.	Canonical Set and Sequence Data Structure	40
27.	Link Instance Diagram	41
28.	Canonical Set Instance Diagram	42
29.	Specification to Design Model Mappings - Aggregate Attributes . .	43
30.	Aggregate Attribute Transformation	44
31.	Specification Symbol to Design Model Mappings - Functional	50
32.	Method Union Algorithm	51
33.	Method Intersect Algorithm	52
34.	Method Difference Algorithm	53
35.	Aggregate Predicate Represented in the Domain AST	55
36.	Aggregate Predicate Represented in the Domain AST after Transform 13	56
37.	Aggregate and Primitive Operation Example	62

Abstract

The feasibility of a transformational formal-based software engineering tool has been the focus of AFIT research for several years. Until now, the main research emphasis has been placed on the individual components that would comprise such a transformational system; therefore, this research demonstrates how a representative collection of aggregate objects would be transformed from specification to code. The research focused on critical integration issues associated with a formal-based software transformation system, such as the source specification, the problem space architecture, design architecture, design transforms, and target software transforms. Software is critical in today's Air Force, yet its specification, design, and development have not achieved a satisfactory level of reliability or consistency. Techniques such as formal-based methods apply sound engineering principles to software development, greatly increasing software's quality and reliability. Because of these important improvements that formal-based methods provide for software development, this research is extremely valuable to the Air Force software development community as a whole.

Transforming Aggregate Object-Oriented Formal Specifications to Code

I. Introduction

1.1 Introduction

Software pervades every aspect of Air Force operations, and has become an essential element of every major Air Force weapon system. In fact, the success or failure of many, if not all, Air Force systems depends directly on their software components. This reliance on software has left the Air Force potentially vulnerable to software's inherent problems, particularly its reliability.

A major reason software has not achieved a satisfactory level of reliability can be traced to its inherent characteristics, primarily its complexity. Every software component of an application has its own uniqueness in that it has a particular functionality, a varying number of states, and it must interact with other system components, both software and hardware.

The domain of a software application also increases its complexity. Software is developed in many different languages, with a number of different tools, on many different platforms. Even the same language has a degree of variance, such as versions and platform-dependent extensions. All of these factors contribute to the dynamic and complicated task of developing software. Because of software's extreme complexity, it must be developed, not manufactured.

Software development is adversely affected by software engineering's relatively short history. As mentioned previously, software is complex and should be developed in a consistent project-based effort. However, software development efforts have been referred to as a cottage industry, where each application is developed in a different, ad-hoc manner. Software complexity, coupled with immature software development techniques, often

results in poorly constructed products containing a number of requirement, design, and implementation or code errors.

Finally, software's high rate of evolution severely impacts its reliability. Because software can be changed easily, a product's functionality during its lifetime can potentially change drastically, making it difficult to ensure a product's completeness and correctness. All of these reliability issues associated with software can be mitigated with effective software methodologies, or strategies to approach software development. One methodology in particular, formal methods, has great potential for increasing the reliability of software.

There are a number of reasons formal methods can be effective in increasing software reliability [7]. First, formal methods force system requirements to be tangible, or concrete. These unambiguous requirements are better understood and are more easily checked for consistency, completeness and correctness. Another positive aspect of formal methods is that they are mathematically based. This allows a system's specifications to be verified and validated using mathematical proofs. Finally, the transform from formal specifications to formal implementations removes many of the implementation's ambiguities that contribute to the system's maintenance complexity.

Development techniques such as formal-based methods apply sound engineering principles to software development, greatly increasing software's quality and reliability. Because of these important improvements that formal-based methods provide to software development, this research is extremely valuable to the Air Force software development community as a whole.

1.2 Problem

Although formal methods have proven their effectiveness in accurately capturing and representing software specifications, there are significant problems that must be resolved before formal methods gain wide acceptance [7]. Major problems identified with formal methods include: software engineers' and end users' lack of familiarity with math-based formal methods; the inherent lack of flexibility formal software specifications have, especially in the problem identification phase; the reluctance management has accepting a methodology

whose benefits have not been clearly established; and the lack of tools to implement these time-consuming, complex formal methods. Although all of the above problems are major impediments to the success of formal methods, none is more critical to its success than the lack of formal based tools.

The feasibility of a formal-based software engineering tool has been the focus of Air Force Institute of Technology (AFIT) research for several years. Research at AFIT has explored tool implementations of two formal methodologies: theory-based model transformation, where the system is represented algebraically, and model-based transformation, where the system retains a formal object-oriented definition throughout the specification and design transformation processes. Considerable work has been done with theory-based model transformations, including two distinct specification-to-design transformation paths developed by Beem [2] and DeLoach [8]. Model-based transformation research has been performed as well, yet not as extensive as theory-based and, therefore, has research opportunities.

Although AFIT's model-based system accepted primitive objects into its domain specification structure, system aspects dealing with aggregate objects needed to be addressed. Furthermore, a design model representation and specification-to-design transformations for model-based systems with aggregates needed to be explored. Finally, the end-to-end verification and validation of a model-based aggregate application had not been attempted. Such a study involved identifying a system with attributes associated with a typical software application, and transforming it through the system. This research focused on critical integration issues associated with a formal-based software transformation system, such as the source specification, problem space architecture, design architecture, design transforms, and target software transforms.

Problem Statement:

Demonstrate the ability of Knowledge-Based Software Engineering (KBSE) technology to accept a specification of an aggregate object, consisting of a collection of primitive object specifications, and successfully transform this aggregate object to code. Resolve critical integration issues associated with a formal-based software transformation system.

1.3 Scope

The scope of this effort was limited to the aggregate object aspects of a model based transform system. The size, complexity, and functionality of aggregate objects used for testing were selected to be sufficient in verifying and validating aggregate object transformation from specification to implementation.

1.4 How This Document is Structured

Chapter 2, *Background*, discusses a variety of software development concepts, focusing on software specification, software design, software implementation, and transformations between specification and design. This chapter also discusses pertinent work previously performed at AFIT. Chapter 3, *The Specification Model*, examines the existing **AFITTOOL** domain model, analyzing major aspects of the model and detailing the modifications made to support aggregation. Chapter 4, *The Design Model*, includes an analysis of a design representation developed by Sward [14], and the extensions made to his model to support aggregation. Chapter 5, *Design Transforms*, describes the aggregate transformations performed within the system. Chapter 6, *Conclusions and Recommendations*, details the findings of this research, and lists areas for future research.

II. Background

2.1 Introduction

The objective of the background research was to gain a basic understanding of pertinent software concepts, particularly overall object-oriented software development methodologies, software specification, both formal and informal, software design, and transformation systems. There is also a section dedicated to related research performed at AFIT. Each topic is discussed in general terms, followed by details specific to aggregate objects.

2.2 Informal Object-Oriented Specification Modeling

Object-oriented development concepts are well-known, so only a quick review will be needed before discussing aggregate specification modeling. Object-oriented methodologies attempt to model a software problem space in terms of real-world objects. There are a number of well-known informal object-oriented software specification modeling techniques, all with their own strengths and weaknesses [7]. In this discussion, Rumbaugh's *Object Modeling Technique*, or OMT, is used as the baseline methodology. All the major modeling techniques attempt to capture the three critical aspects of a software system specification: structural, dynamic, and functional.

Structural aspects of a software specification pertain to the system's entity composition. A system's structure is normally modeled in an object model and outlines an object's attributes and methods.

All software systems have temporal aspects, or how they behave over time and in certain situations. State transition diagrams and event traces are often used to document the dynamic aspects of a software system.

The functional model defines the algorithmic or computational aspects of the application. Data flow diagrams are typically associated with functional modeling.

With these basic object-oriented facts identified, a more detailed examination can be performed on aggregate specification issues, specifically objects with components and objects with relations to other objects.

2.2.1 Informal Aggregate and Association Specification Modeling. Rumbaugh models aggregation as the "part-of" relationship between objects. Aggregate notation consists of one or more classes within the aggregate class, along with multiplicity of each of the component classes.

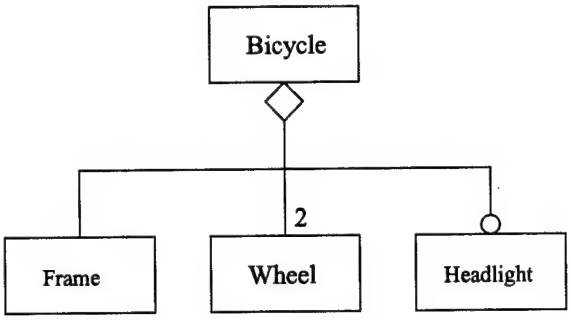


Figure 1 Bicycle Aggregate Example

Looking at Figure 1, for example, the aggregate object *bicycle* is comprised of a single object *frame*, two objects of type *wheel*, and an optional object *headlight*. For a complete listing of aggregate representation options, see [12]. With the exception of its component classes, an aggregate is identical to primitive objects structurally. The complexity of aggregation lies in its integration issues–that is, how components interface and interact within the aggregate. Components in the OMT can communicate with other components or with the aggregate class. Coad and Yourdon's aggregate model is similar to the OMT structurally, but uses object state models to capture dynamic aspects of the specification, and functional aspects of the model are captured in what are defined as service charts, similar to flow charts [5]. An important characteristic to consider is that the aggregate association is a restricted form of an *association*, its major difference being that the association is directional, where components are part of the aggregate.

Associations differ from aggregate relations in that there is no explicit direction or hierarchical relationship in an association. According to Rumbaugh's OMT, an association shows a relationship between two or more objects and is bi-directional. An association can have the following attributes and/or components: two or more classes in the association (typically two); multiplicity of the associated classes; roles of each class in the association; a qualifier, especially in a zero to many association; and ordering or sequences.

In addition, associations can have attributes that only relate to the association, not to the classes in the associations. Finally, associations themselves can be modeled as classes, complete with attributes and operations. Coad and Yourdon's model refers to associations as connections. These connections have multiplicity and optional roles, and any connections with attributes and/or operations are modeled as their own class. Although more restrictive than the OMT, this is a conceptually good approach when considering a modeling technique.

Informal specification modeling techniques such as the OMT are widely accepted in the software community. There are, however, inherent problems associated with these techniques. First, they typically allow ambiguity within the specification, potentially allowing the specifier's intent to be misinterpreted. Also, the correctness and consistency of informal specifications are hard to verify and validate. Because of these two problems, transforming a system from specification to design correctly is difficult at best using informal methods. A possible solution is the use of formal specifications.

2.3 Formal Specification Modeling

Defining a system using formal specifications can mitigate or eliminate problems associated with informal methods. Formal methods are mathematically based, and produce unambiguous specifications. Formal specification modeling can be divided into two separate disciplines: theory-based, where a system is represented algebraically, and model-based, where the system is represented using a formal state-based definition. One of the major model-based specification languages is *Z*, which models objects as schemas and captures aggregate information using sets, sequences, and functions. The following subsections discuss the use of *Z* to define an aggregate object-oriented specification (for more information on primitive *Z* specifications, see [17]).

2.3.1 Model-Based Aggregate and Association Specification Modeling. Single components within a *Z* aggregate are fairly straightforward to understand and model. Declaration of a single component is accomplished by declaring an attribute of some com-

ponent class type. A number of operators apply to single components, such as equality, inequality, and set membership.

Company

$workforce : P\ Employee$
$ceo : Employee$
$chairman : Employee$

$ceo \in workforce$
$ceo \neq chairman$
$chairman \notin workforce$

Figure 2 Single Aggregate Component Example

Figure 2 illustrates how valid single components can be represented in a *Z* schema. Here, an aggregate object *Company* has a set *workforce* and two single components, *ceo* and *chairman*. The *Company* has rules expressed as invariant constraints that the *ceo* must be in the *workforce*, the *ceo* cannot be the *chairman*, and that the *chairman* cannot be part of the *workforce*.

Analyzing component sets is a much harder task. Component set attributes are declared as a power set of a component class. The complexity of a set attribute lies in the large number of operations that can be applied to sets, the number of ways sets can be expressed in predicates and expressions, and the concept of accessing component classes within the set.

Figure 3 extends the *Company* specification example to include *union* and *management* sets of employees. The single components are evaluated as before. The added invariant constraints state that there are no *employees* in both the *union* and in *management*, the members of *management* combined with the members of the *union* comprise the *workforce*, and the number of the members of *management* must be less than 10. The example demonstrates the use of the set operators **union**, **intersection**, and **cardinality**. Other set operators not shown in the example include set **difference**, which returns all elements in set A not found in set B, and the relational operators **subset** ($\subset$) and **subset or equal** ($\subseteq$).

Company

$workforce : P\ Employee$
$union : P\ Employee$
$management : P\ Employee$
$ceo : Employee$
$chairman : Employee$

$ceo \in workforce$
$ceo \neq chairman$
$chairman \notin workforce$
$management \cap union = \{\}$
$workforce = management \cup union$
$\#management < 10$

Figure 3 Component Set Example

Associations build on the functionality of sets, as shown in Figure 4.

Company

$workforce : P\ Employee$
$active : P\ Employee$
$disabled : P\ Employee$
$shifts : P\ Shift$
$days : P\ Day$
$works : (workforce \nrightarrow shifts)$
$daysoff : (days \leftrightarrow active)$

$workforce \setminus \mathrm{dom}\ works = disabled$
$\forall x : \mathrm{ran}\ daysoff \bullet \#\{y : \mathrm{dom}\ daysoff \mid (y, x) \in daysoff\} \leq 2$

Figure 4 Component Association Example

In this example, two associations, *works* and *daysoff*, have been introduced. The association *works*, defined as a partial function, states that *shifts* can have zero or more members of the *workforce* associated, and each member of the *workforce* can belong to zero or one shift. The other association, *daysoff*, is represented as a relation and states that each member of *days* can have zero or more members of the set *active*, and vice-versa. Evaluating the invariant constraints, we find that the set *disabled* is determined by the *workforce* minus the domain of the association *works*. The final invariant constraint is an

example of how to restrict an association with invariant constraints, in this case restricting the domain of an association. This invariant constraint translates that all active employees can have two or less days off.

2.3.2 Formal Functional Specification Modeling. There are a number of *Z* predicates and expressions that can be used to express pre and post conditions in *Z* dynamic schemas [3] [17] [13]. The two main predicates that are associated with aggregate schemas are **Universal** and **Existential** quantification. **Universal** quantification states that the predicate and its associated expressions apply to every element within a specified set. An **Existential** predicate states that there exists at least one element that meets the predicate and expression constraints, also within a specified set. Figure 5 illustrates a dynamic schema for the functional model that uses an **Existential** predicate.

$$
\begin{array}{|l}
\hline
ChangeShifts \\
\Delta Company \\
worker? : Employee \\
newShift? : Shift \\
\hline
\exists x : workforce \bullet x = worker? \land x \in \mathrm{dom}\, works \\
works' = \{works \setminus works \rhd \{worker?\}\} \\
\cup(worker?, newShift?) \\
\hline
\end{array}
$$

Figure 5 Functional Schema Example

In the example, the functional schema *ChangeShifts* includes the object Δ *Company* and has the inputs *worker?* and *newShift?*. An existential predicate is used as a single precondition, verifying that the input *worker?* exists in both *Company.workforce* and in the domain of the association *Company.works*. The post condition changes *Company.works* by removing the old association of *worker?* and adding the new association with inputs *worker?* and *newShift?*.

2.3.3 Formal Dynamic Specification Modeling. *Z* is well-suited for defining structural and functional aspects of a specification. *Z* does not provide a structure to represent the dynamic model. Therefore, this research looks at the use of State Transition Tables

(STTs) to capture dynamic aspects of a software specification. System states and their transitions are key aspects of the aggregate dynamic specification and are formally captured at the class level as an STT entry. Their structure is as follows:

Current	Event	Guard	Next	Action	Send

Current represents the state in which the class currently resides. The *Event* field identifies what event or events the class can react to in that state. An optional *Guard* or constraint can be defined over the transition. The state to which the object can transition is identified in the *Next* field. Any operations that occur on transition are captured in the *Action* field. Finally, the *Send* field identifies any messages generated as a result of a transition. To illustrate how an STT can be populated, a simple *Intersection* domain will be used as an example (see Figure 6) . The *Intersection* domain identified in Figure 6 consists of a *TrafficLight*, a *Timer*, and a traffic *Detector* that detects traffic waiting at the traffic light.

TrafficLight

Current	Event	Guard	Next	Action	Send
MainGreen	Car_Detected		MainYellow	update_1	Set_Timer
MainYellow	Timer_Expires		SideGreen	update_2	Set_Timer
SideGreen	Timer_Expires		SideYellow	update_3	Set_Timer
SideYellow	Timer_Expires		MainGreen	update_4	

Timer

Current	Event	Guard	Next	Action	Send
Idle	Set_Timer		Timing	set_time_left	
Timing		$time\_left = 0$	Idle		Timer_Expires

Detector

Current	Event	Guard	Next	Action	Send
Idle	Arrival		Idle	ItemArrival	

Figure 6 State Transition Table Example

There are three key event scenarios among classes that must be resolved. The first is where an object sends a message that another object is expecting, and the event names match. In the example above, the *TrafficLight* object sends out a *Set_Timer* event that the timer is expecting. Here, the events only need to be validated as a correct match.

The second scenario exists where an object sends a message that another object is expecting, but the event names don't match. Again, looking at Figure 6, the reusable primitive object *Detector* sends the event ItemArrival that corresponds to the *TrafficLight*'s *Car_Detected* event. This disparity must be detected and resolved for the STT to be valid.

A third scenario exists where an object sends a message that matches another object's event name, but is coincidental. An example would be where a car has components *Engine* and *Air Conditioning*, and they both have a *Turn_Off* event. Again, events must be matched within the problem space for the STT to be valid.

Event matching brings up other situations that must be addressed such as how send events without receivers are handled, and how expected events without a sender are resolved.

2.3.4 Restricting Z Specifications Using Declarations and Invariant Constraints. As with any specification representation, there are many ways in Z and the OMT to capture or model an aspect of a system specification. Although this extensive set of representations gives the specifier flexibility, it makes any effort to automate specification transformation exceedingly difficult. Therefore, a primary goal of this effort was to find a single representation to model constraints in a specification. There must also be some criteria for selecting a single representation. First, the representation methods had to have relevence to aggregate classes. Second, the methods for specification had to be equivalent. If the representations met these criteria, the most succinct representation was selected. (NOTE: This was not always the case–the Z parser implemented by [17] in some cases could not parse the most succinct representation. In this case, a parsable equivalent was chosen) To illustrate this point, the *shop stock* example from [3] will be used. In the example, a shop has a number of different items currently in its inventory, and is modeled as follows:

Inventory

$$stock : P(ITEM \times \mathcal{N})$$

$$\forall i : ITEM;\ m, n : \mathcal{N} \bullet (i, m) \in stock \land (i, n) \in stock \Rightarrow m = n$$

Or, as a set former expression:

$\{s: (ITEM \times \mathcal{N}) \mid \forall\, i: ITEM; m,n: \mathcal{N} \bullet (i,m) \in s \wedge (i,n) \in s \Rightarrow m = n\}$

stock is a power set of all items and an associated natural number, or quantity. The type *Natural* ($\mathcal{N}$) is a valid constraint placed on quantity, in that it is impossible to have a negative number of any item. Had the attribute *stock* been left as it had been declared, it would have been possible to have the same item have multiple quantities, which is also impossible in this scenario. A Universal predicate is therefore written as an invariant constraint that precludes a single item from having multiple quantities.

Although this is a correct way to specify this relationship, it would be less verbose to constrain the relationship at declaration. A more succinct way to capture this specification is by establishing a functional relationship.

Inventory

$stock : (ITEM \nrightarrow \mathcal{N})$

A partial function by definition restricts any *ITEM* from having more than one quantity, yet many ITEMs can have the same quantity. There are a number of other relationships that can be expressed in *Z*, as shown in Figure 7 .

	Domain	Range	Specification	Symbol	Rumbaugh	LaTeX
m:n	either	either	Relation	A ↔ B	A ●——● B	`\rel`
n:1	optional	optional	Partial Function	A ⇸ B	A ●——o B	`\pfun`
n:1	required	optional	Total Function	A → B	A ●—— B	`\fun`
n:1	optional	required	Partial Surjection	A ⤀ B	A 1+——o B	`\psurj`
n:1	required	required	Total Surjection	A ↠ B	A 1+—— B	`\surj`
1:1	optional	optional	Partial Injection	A ⤔ B	A o——o B	`\pinj`
(alternate partial injection)				A >⇸ B		`>\!\pfun`
1:1	required	optional	Total Injection	A ↣ B	A o—— B	`\inj`
(alternate total injection)				A >→ B		`>\!\fun`
1:1	required	required	Bijection	A ⤖ B	A —— B	`\bij`
(alternate bijection)				A >↠ B		`>\!\surj`

Figure 7 Association Multiplicity

Figure 7 outlines the types of relations that can be used, as well as their domain and range multiplicities, the symbols used, corresponding Rumbaugh notation, etc. What should be apparent in these definitions is that each association has its own level of restriction that can be applied to a specification. Figure 8 is a Venn diagram illustrating how each relation corresponds to the others.

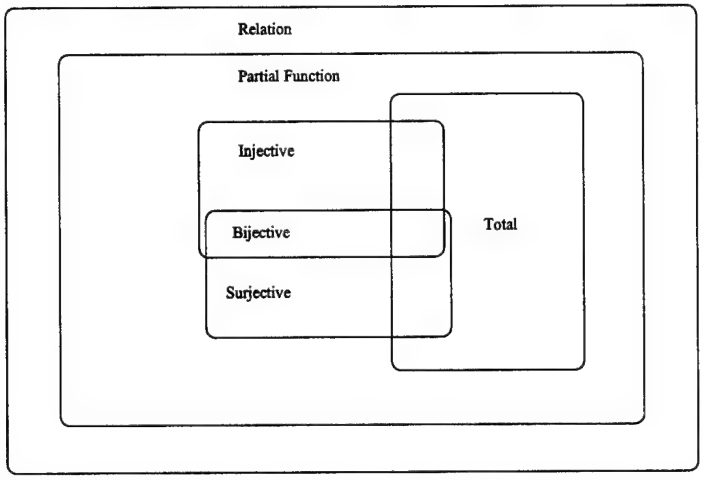


Figure 8 Function Venn Diagram

2.4 Software Design

There are a number of software design methodologies, yet they all share common goals [12] [6] [7]. When constructing a software system's specification, the main goal is to capture "what" should be in the system. The design of the system, on the other hand, should focus on the "how". With this in mind, a design model should be able represent how a system should be designed, from choosing class and data structures to defining algorithms. The design model structure should apply software engineering principles to any design it represents. Object-oriented concepts such as inheritance and polymorphism should also be supported. It should also be at a high enough level of abstraction so that implementation traits are not implied.

2.5 Relationship Between Software Specification and Design

Dependencies between specification, design, and implementation aspects of software development are inevitable. Swartout and Balzer contend that most if not all specifications are embedded with implementation constraints [15]. Examples of implementation constraints within the specification include a target language, hardware platform, methodology, or environment. Implementation constraints applied in the specification restrict the number of design options available in the design phase.

Embedding implementation constraints within the specification is not always detrimental to a software development effort. On the contrary, many times it is necessary to scope the effort, or to capture known constraints. Swartout and Balzer point out that specifications void of implementation restrictions would force the potentially daunting task of reviewing all possible implementation technologies.

A far worse scenario involves misunderstood or absent implementation constraints in the software specification. This situation ultimately results in specifications that cannot be implemented as specified. At best, the specification must be revisited to evaluate the constraint, usually a time-consuming process. At worst, the final application is developed and does not implement the system as specified.

2.6 Software Development Transformation

All software development efforts go through a series of transformations. Specifications, for example, must be transformed to a corresponding design representation and the design representation must be transformed to an implementation. To accomplish a transformation, then, the source set of entities in a transformation must be mapped to a target set of entities [4]. Transformations can be classified as: equivalence, where information is neither loss nor gained; information-losing, where the transformation's target is less constrained than the source; and information-gaining, which is the inverse of information-losing.

Transformations within software systems can be grouped into the three object-oriented modeling categories. There has been considerable work done with the object

or static model transformation, and implementations exist in a number of CASE tools and database products [4]. Functional and dynamic model transformations have also been studied, but not at the level that structural transformation has achieved.

2.6.1 Aggregate and Association Structure Transformation. A designer has a number of options when respresenting the structural aspects of aggregates and associations in the design and, ultimately, implementation. For instance, the designer could create a container class to encapsulate components. Another option is to embed one-way or two-way pointers within the component classes. Still another option is to combine the two classes. All these design decisions have benefits and drawbacks. Creating container classes, for instance, achieves maximum reusability because the component classes are not altered. The drawback to using container classes is that they are inefficient for even moderately large containers. The point here is to allow the designer to have the flexibility to choose and test each design decision.

2.7 Categorizing Transformations

The first aspect of examining an end-to-end transformation system is to categorize what type of transformations can occur in the system. For this effort, all transformations within the system are categorized as either inter-model or intra-model.

Inter-model, or model-to-model transformations, map a source entity to a physically or logically distinct target entity. To put it another way, inter-model transformations provide the interface from one representation to another. After examining the transformation system **AFITTOOL**, for example, there are three distinct interface points in the system: from the *Z* specification to the specification domain, from the specification domain to the design domain, and from the design domain to the target implementation. Ideally, inter-model transformations perform straightforward, one-to-one mappings from domain to domain, with little or no user input. In fact, the *Z* system specifications are parsed (transformed) into the specification domain and the Ada implementation code from the design model is generated using context-free grammars, which require no user input. The primary reason why these direct mappings between models are advantageous is to maintain

the integrity of each entity in the system. For example, the design modeler should be able to transform the specification into a design representation without being forced to make specification decisions during the process. Unfortunately, a complete one-to-one mapping is exceedingly difficult to achieve between models. The problem lies in the fact that these transformations must map equivalent structures between two vastly different models. In many cases, transformations must occur within the model itself to prepare it for inter-model transformations. These types of transformations are categorized as intra-model.

Intra-model transformations are refinements performed within a model. For the purposes of this research, intra-model transformations defined occur in both the specification and design domains. Intra-model transformations can be automatic, such as converting a program construct to a more canonical representation, or user-assisted, such as selecting a type for a previously undefined attribute. The main goal when performing intra-model transformations is to refine the model's structure in preparation for transformation to the next stage.

Overall, there are a number of issues involved in transforming aggregate components and associations. First, component objects are represented in the specification model as single entities, sets, or sequences. Since these representations are not typically found in implementation languages, they must somehow be transformed. Aggregates and associations also have associated multiplicities that must also be transformed. As with primitive objects, invariant constraints are applied to aggregate and associative declarations, and since invariant constraints do not have a direct design counterpart, they, too, must be transformed. Aggregate STTs capture aggregate and component level events and transitions, and the event scenarios identified earlier in the chapter must be resolved and transformed. Finally, there are operations unique to associations and aggregates that must be transformed.

2.8 Assessment of Previous AFIT Research

The feasibility of a formal-based software engineering tool has been the focus of AFIT research for several years. First, a typical transformation architecture was chosen to represent the system and its components (See Figures 9 and 10). Because of its wide acceptance

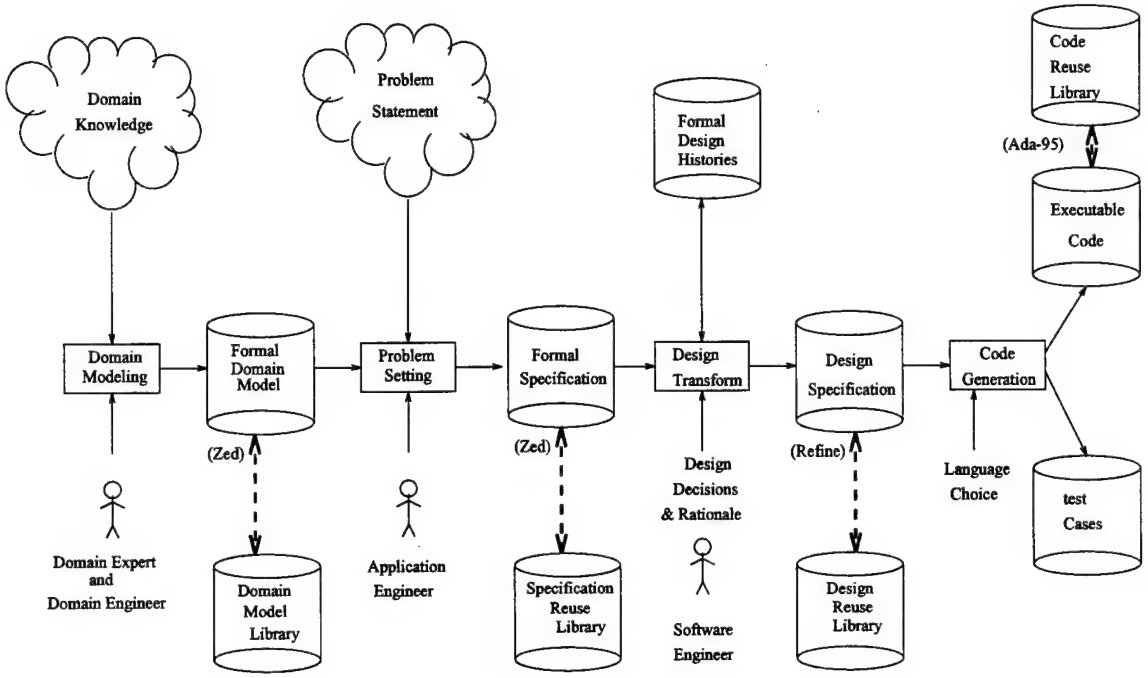


Figure 9 Software Transformation System

in the software community for representing software systems, the decision was made to use object-oriented modeling methods. The decision to use Rumbaugh's Object Modeling Technique (OMT) as the modeling methodology was based on the OMT's relative maturity [12]. Although object oriented approaches are an accepted method to represent software systems, they are informal. Therefore, a representation in the transform system was needed to decompose these objects into a consistent, general format. Abstract syntax trees (ASTs) were chosen to provide both the domain model and design representation structures in their canonical forms. (See Figure 9) With these underlying structural issues resolved, Hartrum and Bailor developed the tool's foundation by integrating the formal method Z with Rumbaugh's OMT, creating a general object-oriented domain model [9]. Wabiszewski demonstrated the ability to generate domain ASTs from Z models by defining an object model AST and developing an AST-building parser [17]. Similarly, Lin developed a system that parses Larch algebraic specifications into a Larch theory-based AST [10]. Beem's work continued Lin's by demonstrating the ability to transform Lin's Larch AST to an algebraic-based model AST developed by DeLoach, called O-SLANG [2] [8]. Mean-

while, the back end of the tool has been fleshed out by Sward's work reverse-engineering FORTRAN code into the Generic Object Model(GOM) and a design-specific AST [14].

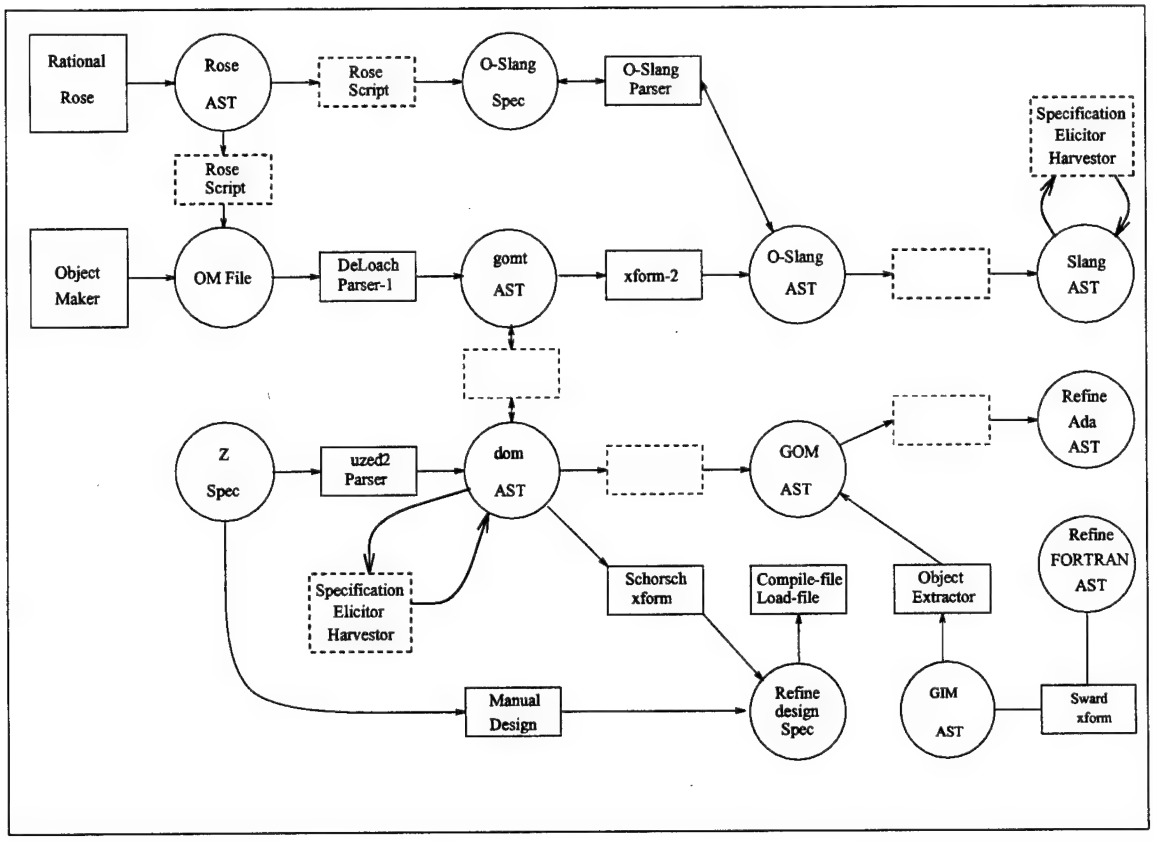


Figure 10 AFITTOOL Component Layout

III. The Specification Model

3.1 Introduction

The first phase of the specification transformation process is to populate a specification domain model from *Z* specifications. There were key assumptions that had to be considered when developing the domain model. First, it was critical that the specification domain effectively and correctly represent an aggregate object's structure, behavior, and functionality. Rumbaugh's OMT, although informal, provides a rich set of modeling techniques, and was therefore a basis for the domain structure. Another prime consideration in creating the specification model was the input specification language, in this case *Z*. Although *Z* provides a formal software representation, in many instances it does not map directly to OMT constructs, and vice-versa. Because of this, where there were *Z* and OMT components that directly correlated, the *Z* component was converted to its OMT equivalent. In all other instances, the *Z* components had to be maintained in their original format. Finally, extensive work has been performed creating a model-based domain structure, especially at the primitive object level. It was determined that the existing domain provided a solid foundation in which to add aggregate components.

3.2 The Existing Primitive Specification Domain Model

The original specification domain model was an abstract syntax tree (AST) composed predominately of two previous AFIT efforts: DeLoach's work developing a generic object model, and Wabiszewski's work parsing *Z* specifications into an AST structure [8] [17]. This section discusses the domain model in its structural, functional, and dynamic aspects.

3.2.1 Domain Model Structure. The way the domain model represents structural aspects of the specification draws heavily on how DeLoach defined objects in his Generic OMT, or GOMT. It does, however, have notable differences, and major components are therefore defined in this chapter. See Figure 11.

Domain Theory. At the highest level, a Domain theory is defined for an entire specification system (see Figure 12). A Domain theory consists of sets of pre-defined types, global constants, global types, and primitive classes.

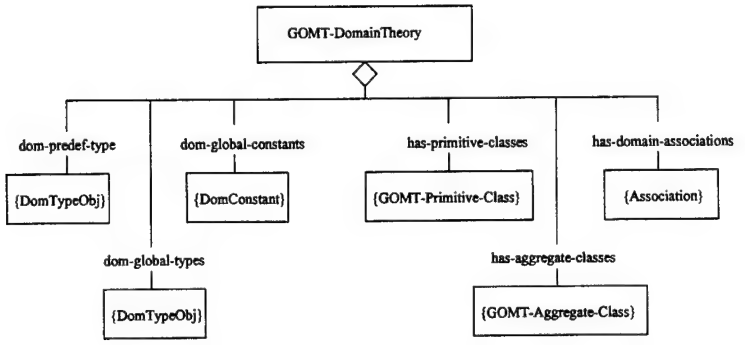


Figure 11 Domain Structure

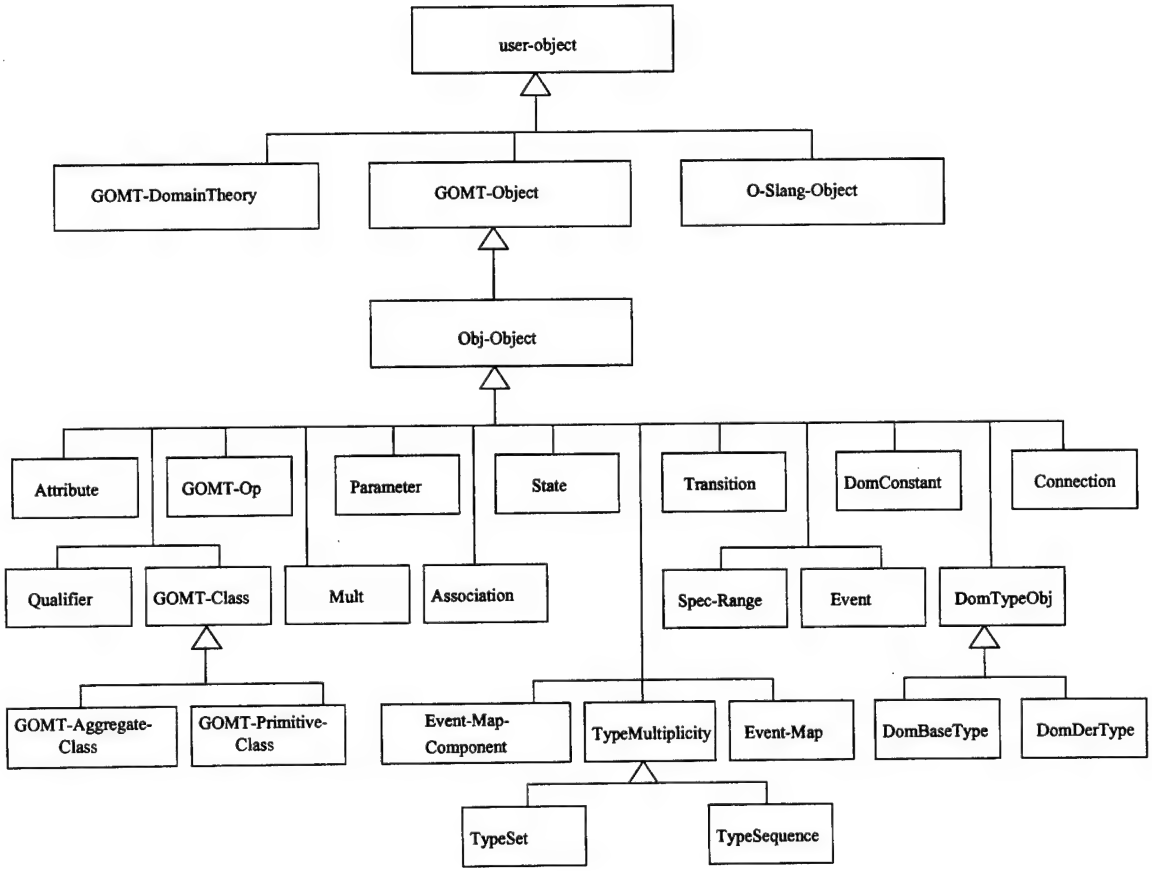


Figure 12 Domain Inheritance Hierarchy

Domain Types. Scalar and composite attribute types are represented in the domain using a structure that can either be a base or a derived type. Base types represent the most common and frequently used types, such as integer, real, and character, but can also represent user-defined types.

$$ALL\_MODE ::= up \mid down \mid hold \mid off$$

$$\begin{array}{|l} COUNT\_MODE : P\, ALL\_MODE \\ \hline \forall x : COUNT\_MODE \bullet ((x = up) \vee (x = down)) \end{array}$$

Figure 13 Domain Derived Type Example

Derived types are more elaborate in that they can have predicates written over them and can represent a set or sequence of types. Figure 13 is an example of a derived type called *COUNT_MODE*, which is a subtype of another type *ALL_MODE*. A universal predicate is written over *COUNT_MODE* that states all members of *COUNT_MODE* must have the enumerated value of *up* or *down*. Finally, derived types have an associated name and can have a sequence of values.

Primitive Class. Each component specification (i.e., specifications without aggregate attributes) is represented in the domain model using primitive class objects. Each primitive class object is composed of the class's name, a flag to indicate whether the class is concrete or abstract, and, if any exist, the class's superclasses, private types, private constants, attributes, invariant constraints, operations, states, events, and transitions.

Primitive Class Attributes and Constants. Primitive class attributes and constants, both global and class-level, have similar structures: all have an associated name, data type, and a value. The obvious difference is that the value of an attribute is optional (default value), where a constant value is required.

3.2.2 Functional Aspects of the Domain Model. The functional model is represented by the operation structure *GOMT-Ops* defined in the primitive class.

Operations. Each Operation has a name and can have associated with it a sequence of parameters, a set of predicates, and a set of sub-operations. These sub-operations allow an operation to be functionally decomposed, if desired.

Predicates. The predicate structure, excluding minor changes, is the same structure defined in [17]. Predicate types important to this research that can be represented include relational, universal and existential quantifications, and implication.

Expressions. Expressions are also defined using [17]. The expressions explored in this research include: cardinality; set union, difference, intersect, and comprehension; sequence concatenation; function invocations; and component attribute references.

3.2.3 Dynamic Aspects of the Domain Model. The dynamic or temporal aspects of the domain model are captured with the entities **Events**, **States**, and **Transitions**.

Events. Events have an associated name and can have a sequence of parameters and a set of predicates.

Transitions. Transitions represent a single entry from a state transition table. They have a caused-by event, a transition-from and a transition-to state, a single action in the form of an operation, and can have a set of send events.

States. States have an associated name and can include a set of predicates and a set of sub-states.

3.3 Domain Model Additions to Support Aggregation

The existing domain model was constructed to support primitive objects, and these elements specific to aggregate objects needed to be incorporated. A review of the existing domain structure revealed that additions would have to be made to represent aggregate object and dynamic aspects of a specification. The aggregate functional model is captured in the existing predicate structure, and no additions were needed.

After a preliminary analysis, the conclusion was reached that aggregate class structures should reside directly subordinate to the domain theory, at the same level as primitive

classes. Also, aggregate classes should extend existing primitive classes, adding the ability to model component classes, associations, and component-level event mapping.

3.3.1 Class Relationship Structure. The first aggregate issue addressed was how relationships between classes , either as a component or as an association, could be modeled. Deloach's Connection structure was used as a basis for capturing relationships between classes (see Figure 14) because it provided the means to represent every attribute of an OMT relationship. Although the connection structure has extraneous components in the aggregate context, it allows a more flexible implementation. A connection is comprised of the maps *has-name*, which refers to the attribute name declaring the connection; *has-class-name*, which identifies the class of the attribute; *has-qualifier*, which is comprised of a name and a datatype; *has-role*, which stores the role in the relationship (e.g., a teacher teaches a class, a class is taught by a teacher); *is-ordered*, used to differentiate between sets and sequences; and *has-mult*, which captures the class's multiplicity. Multiplicity can be represented the following ways: one, many (zero to many), plus (with an integer to capture 1+, 2+, etc.), optional (zero or one), and specified with a range. A separate map from connection to a class reference is created and maintained for quick reference. Upward multiplicity in a Connection is captured implicitly.

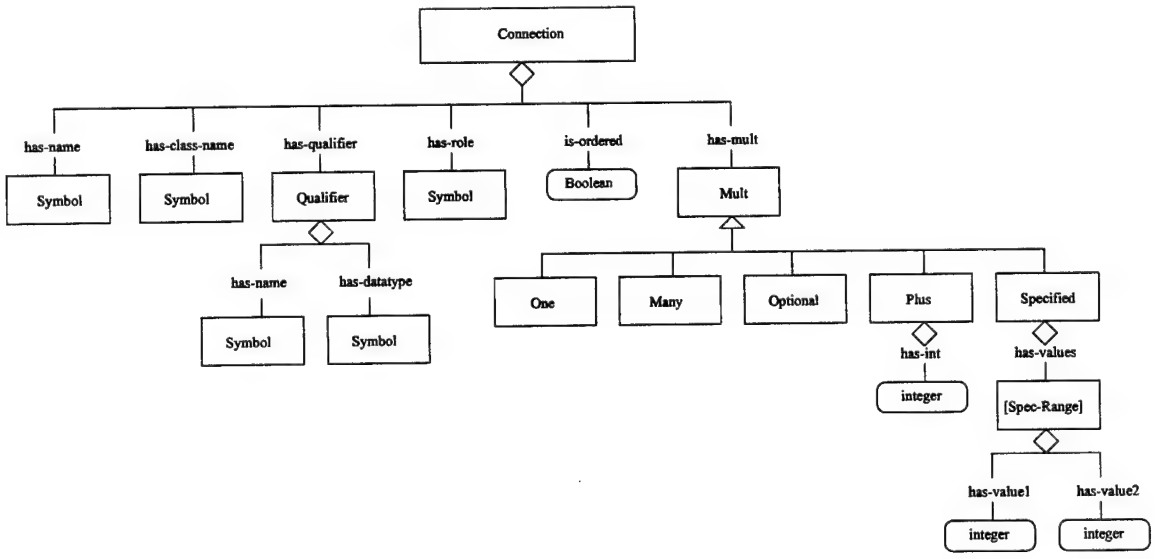


Figure 14 Connection Structure

3.3.2 Aggregate Class Structure. A structure was needed to represent components of an aggregate class. Modeling the aggregation relationship independent of the aggregate and component objects, much the same as an association, is another approach, but does not intuitively make sense–aggregates are not aggregates without their components. Deloach's aggregate model using the connection structure provides a solution (see Figure 15).

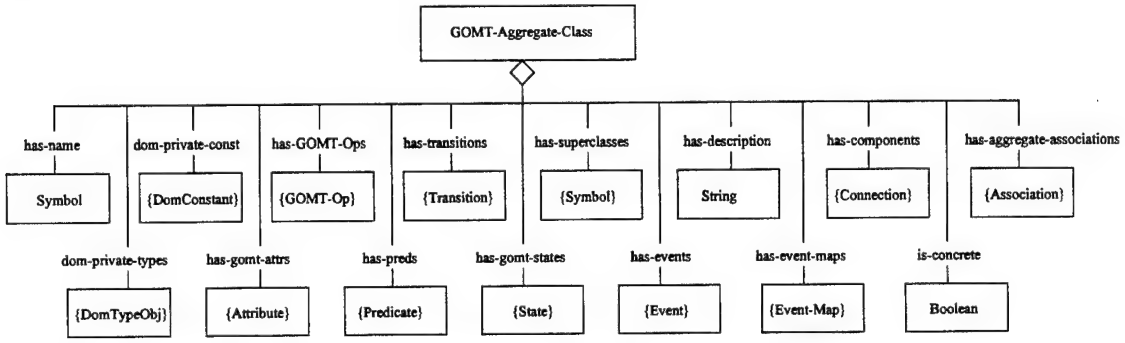


Figure 15 Aggregate Class Structure

3.3.3 Association Structure. Deloach's modeling of associations was used as the basis for the domain model representation. Associations exist at both the class level and subordinate to aggregate classes. An association is comprised of an association name and a set of two or more connections. The association also includes an optional **GOMT-CLASS** to represent associative objects (see Figure 16). Multiple components are currently allowed in an association. However, this effort does not evaluate allowing more than two objects within an association. This may be too complex to evaluate in a design transform.

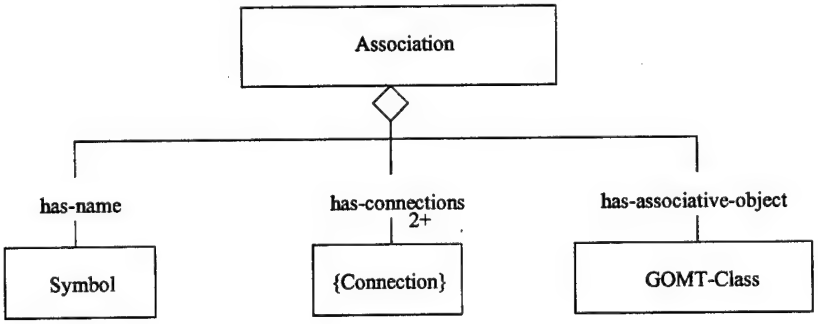


Figure 16 Association Class Structure

3.3.4 Event Map Structure. The event map structure was created to capture all component event maps. Each event map consists of an event sender component and one or more event receiver components. Event maps are subordinate to aggregate classes.

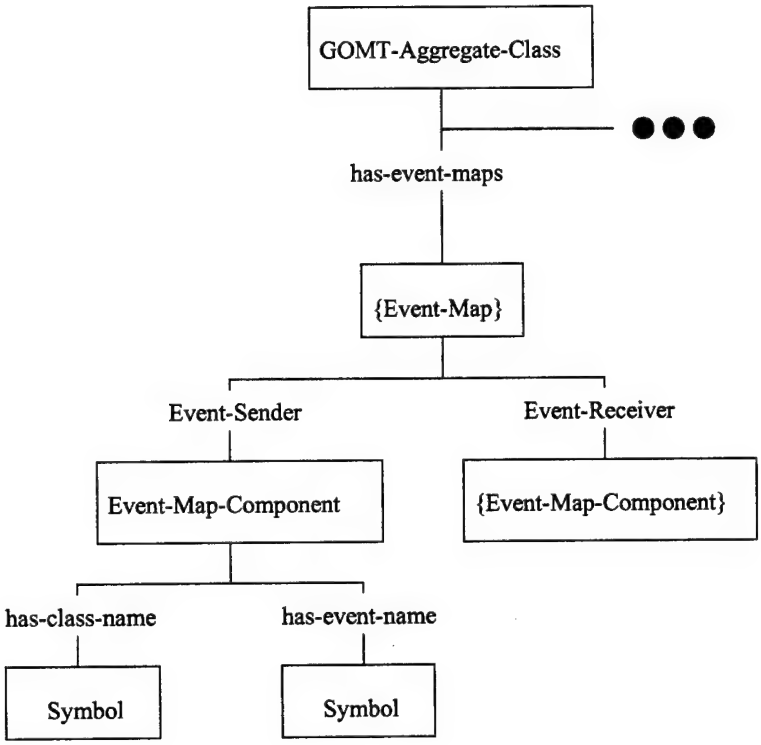


Figure 17 Event Mapping Structure

IV. The Design Model

4.1 Introduction

This section describes how the design model is constructed, as well as how aggregate components are represented within the design model.

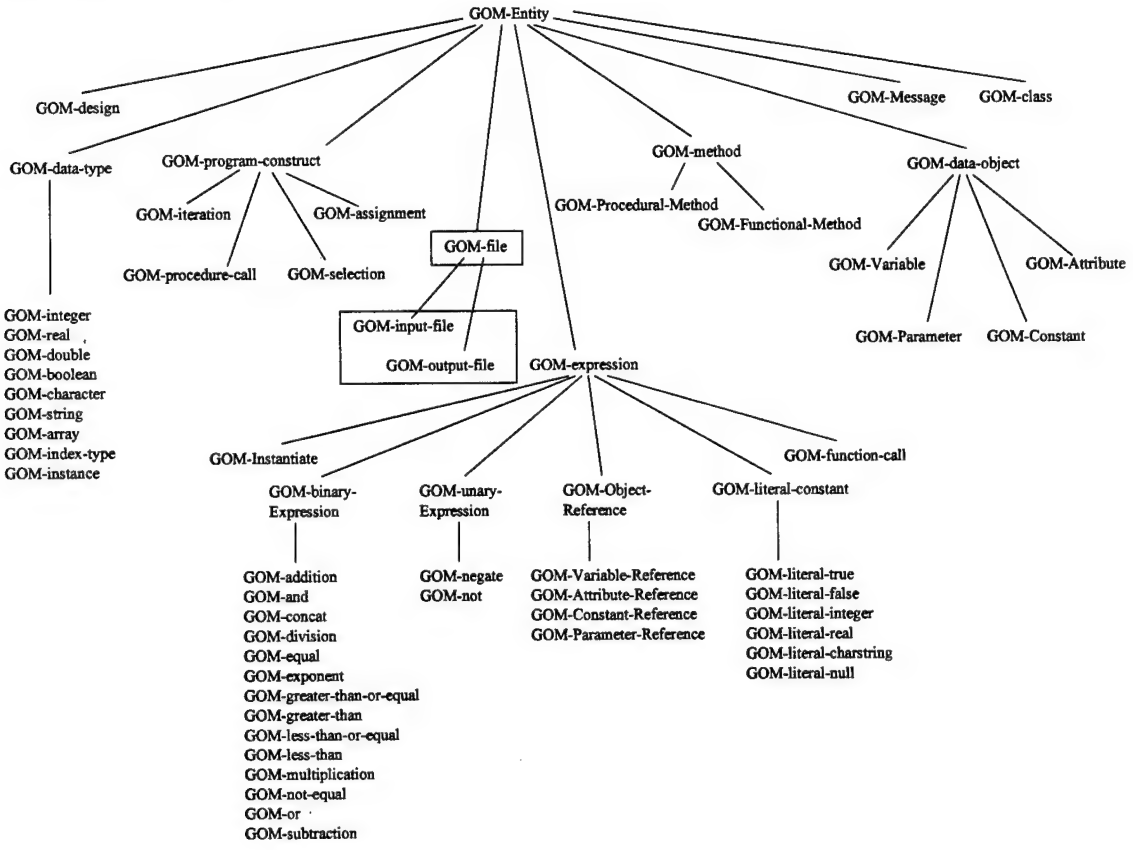


Figure 18 Design Inheritance Hierarchy

4.2 Analysis

Sward developed as part of his research a canonical design model called the Generic Object Model, or GOM [14]. Sward's GOM provided a sound preliminary model that was modified and extended in this research, particularly because the GOM has the ability to represent basic object-oriented components, including object classes, attributes, constants, and methods (see Figures 18 and 19).

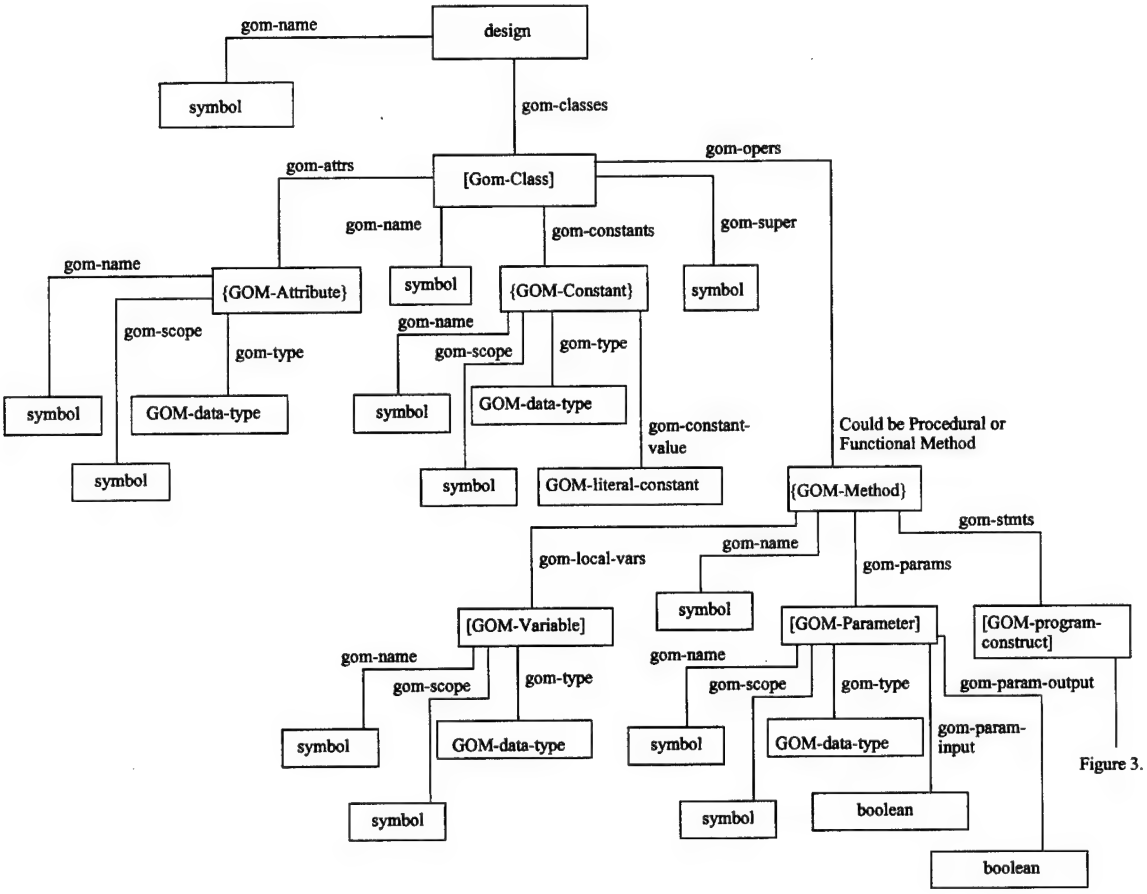


Figure 19 Design Structure

4.2.1 The GOM. The GOM, developed using a Refine AST, represents a system's object-oriented design. At its highest level, a design has a name and consists of a sequence of entities called Gom-Classes. Each Gom-Class has an associated name, set of Attributes, set of constants, set of operations, and a single reference to a superclass. Although other object-oriented design methodologies allow for multiple inheritance, this feature would substantially increase the model's complexity.

Each attribute within a class has an associated name, scope, and data type. Sward defines scope as the visibility of an entity, which can hold the values *public* or *private*, and is also associated with a class's attributes, constants, variables, and parameters. Data types defined within the GOM are integer, real, double, boolean, character, and string.

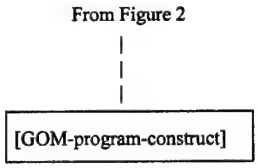
The composition of constants within the GOM include a name, scope, data type, and literal values. Valid literal constant values allowed in the GOM are *true*, *false*, *integer*, *real*, *charstring*, and *null*.

A method in the GOM is comprised of a name, a sequence of variables, a sequence of parameters, and a sequence of program constructs. A variable has the identical structure of an attribute, as defined above. A parameter extends the variable structure by adding two boolean values, one identifying the parameter as an input, the other as an output.

Program constructs can assume three basic structures: *assignment*, *selection*, and *iteration* (see Figure 20). An assignment construct consists of a *GOM-object-reference* as the left-hand side of a construct, and a *GOM-expression* on the right-hand side. Object references in this context are references to GOM variables, attributes, constants, and parameters. An expression can be either an instantiation, a binary expression, a unary expression, an object reference, a literal constant, or a function call. Binary expressions supported include *addition*, *and*, *concat*, *division*, *equal*, *exponent*, *greater-than-or-equal*, *greater-than*, *less-than-or-equal*, *less-than*, *multiplication*, *not-equal*, *or*, and *subtraction*. Two unary expressions are supported: *negate* and *not*.

The selection construct (eg, If-Then-Else) consists of a *GOM-expression*, a sequence of program constructs that represent the "then" part of the selection, and a sequence of program constructs that represent the "else" part of the selection.

Finally, the iteration construct (eg, While..Do) is composed of a *GOM-expression* to represent the "while" portion of the construct, and a sequence of program constructs to represent the iteration body.



A GOM-program-construct could be a GOM-assignment, GOM-selection, or GOM-iteration. All of these constructs are shown below:

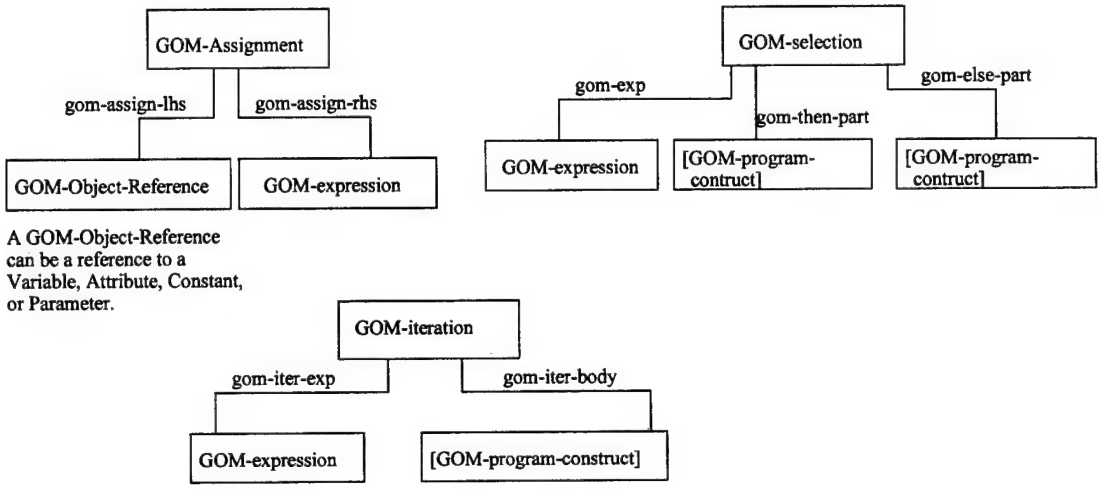


Figure 20 Design Program Constructs

4.2.2 Representing Aggregation in the GOM. When talking about associations and aggregates in a generic object-oriented design model, there must be the concept of object class referencing. In a real-world implementation, references to other class-like entities are captured in the form of pointers, subscripts, keyfields, etc. In Sward's GOM, a reference type did not exist and has to be included.

Data structures in the domain model are represented as sets and sequences, where any restrictions to these structures are applied using invariant constraints. Although sets and sequences are excellent at abstractly representing these compound data structures, there are few widely-used, wide-spectrum languages that support their implementation. Sets and sequences, therefore, must eventually be converted to more common data structures.

Representing sets and sequences of object classes in the design model could be accomplished a number of different ways. As stated earlier, any representation should support incremental transformations to ensure maximum flexibility when executing design decisions. The two most prominent choices considered were to extend the design model's operation set to include set and sequence operators, or to include the operations as methods of a canonical set or sequence class.

Extending the design model's operation set to include set and sequence operators would allow straight transforms from the domain model to the design model, effectively deferring their transforms and allowing the design model to remain highly abstract. However, this approach further complicates the design tree structure with the addition of set and sequence operators. Also, since these operators must eventually be converted to more typical program constructs, this approach only defers the inevitable transformation of sets and sequences.

Including the operations as methods of a canonical set or sequence class is a more flexible solution to the problem, as it adds no structural complexity to the design model yet fully supports all set and sequence operators. Set and sequence operator concepts are maintained as methods, which can also be transformationally refined. The major drawback to this approach is that the structure is"soft"; that is, the design is maintained programmatically, not structurally.

4.3 Extensions to the GOM

The following extensions were made to the GOM in support of aggregate classes. It is important here to note that extensive changes were made to the GOM in support of both primitive and aggregate classes. For a summary of the basic changes, see [16].

Add reference types. Without reference types, aggregate class designs could not be represented.

Capture set, sequence, and association operators as methods. Because of its flexibility, this model captures all set, sequence, and association operators as methods in canonical classes.

Add the ability to represent generic classes. As stated previously, set, sequence, and association operators are captured as methods in canonical classes. These canonical classes provide the same functionality, with the only variance between the same type canonical classes being the actual class it contains. It makes sense, then, to represent these canonical classes as generic entities.

Add public and private entities. A private attribute of type **boolean** was added to the design AST at the GOM-Entity level. This allows all affected components to be set public or private, as appropriate. Components that do not have a public or private attribute associated, such as GOM-Expressions, simply do not use the attribute, and it is ignored in transformations. Many object-oriented languages such as Ada and C++ support multiple levels of privacy. However, the number and level of restriction varies greatly from language to language, and the boolean attribute was sufficient to capture the accessibility of these entities.

V. Design Transforms

5.1 Introduction

In the previous two chapters, the structures of both the specification and design models were defined. Two goals were used when developing these models: the first goal was to ensure the models could support the methodology used to transform the aggregate objects. The second goal was to ensure all pertinent specification and design aspects of a system were captured in an abstract or canonical format, maintaining each model's completeness and correctness. This chapter provides the underlying assumptions used in developing the complete end-to-end transformation of specifications. Also, it outlines the system-level design decisions used in defining the transformation system, and the overall strategy used to transform the aggregate and associative specifications to their respective design representations. Finally, it identifies what aggregate transformations occur within the system.

5.2 Transformation Assumptions

As with any research effort, the research must be conducted with assumptions. These assumptions were made to scope the effort. There has been a varying amount of work on a number of these assumptions and the extent of that work is documented accordingly.

ASSUMPTION 1: All *Z* specifications parsed into the domain model are correct. The specification domain is populated using a *Z* parser, which catches several types of specification errors. However, there are a number of errors that can be introduced into the specification domain, such as inter-object referencing and object type inconsistencies. It is therefore assumed that the *Z* specifications populating the specification domain are free of errors.

ASSUMPTION 2: All associations are binary. The specification domain does allow n-ary associations to be modeled. However, this effort has been conducted using *Z* relations, whichare inherently binary. According to Rumbaugh, this is not much of a limiting factor, as most associations are binary [12]. The specification model does support associative objects, which, properly modeled, can represent a limited ternary association.

ASSUMPTION 3: Dynamic model design aspects of aggregate objects are not addressed. The dynamic aspects of the specification are represented in the specification domain, including an aggregate event map structure that was defined in Chapter 2.

5.3 System Level Decisions

This section captures decisions made in the definition of the aggregate transformation system, including the rationale for each decision. A number of these decisions were needed to maintain consistency within the design domain. Finally, all decisions having an impact on primitive objects within the domain were closely coordinated with Tankersley [16], the developer of the primitive object transforms.

Decision 1. Aggregate components and associations can either be represented as actual instances of component objects, or as references to instances. For consistency, this methodology transforms all component and association declarations to references.

Decision 2. A decision was needed to determine what programming methodology the design model should support. A design model accommodating a wide-spectrum representation would provide the most flexibility, yet there are few implementation languages that are wide-spectrum. A design model supporting a structured methodology has numerous target implementation languages, yet places a number of restrictions on the design. The decision was made to choose a design model based on the object-oriented methodology, because it allows the design to be represented with a high level of abstraction, and because there are a number of languages that support a object-oriented methodology.

Decision 3. A target implementation language needed to be selected for the transformation system. The fact that the design model is object-oriented limited the implementation to languages that support object-oriented constructs, specifically C++, Java, and Ada. From these choices Ada was chosen because the language's strong typing and rigorous program correctness checking at compilation and link time facilitated the generated code's validation.

Decision 4. How sets, sequences, associations, and their associated operations could be represented in the design model was briefly discussed in Chapter 4. The decision

was made to capture aggregate aspects of the specification using the container approach because containers can be represented with a high level of abstraction, yet can be directly implemented.

Decision 5. The container classes used to represent aggregate attributes have very similar structure and functionality. Therefore, the methodology incorporates the concept of generic classes to represent containers. Although a generic class is an Ada-specific construct, other OOP languages support similar constructs. C++, for example, has a generic class equivalent called *templates.*

Decision 6. An overall strategy was needed to implement set, sequence, and association operators. The object-oriented methodology used to represent the design model does not support these types of operators, nor were there object-oriented operators that were equivalent to set, sequence, and association operators. The decision was therefore made to represent set, sequence, and association operators as methods of new object classes.

Decision 7. The ability to produce or generate an implementation representation from the design model was needed. Ada grammars for both specifications and bodies were created from the design model to create the Ada implementation surface syntax.

Decision 8. The use of pure functions in a design model may be required, yet they don't necessarily apply to any one design class. To capture pure functions, then, a strategy of creating an operation library primitive class to house these functions was used.

Decision 9. Aggregate set, sequence, and association attributes have *Add* and *Remove* methods defined in the aggregate class. This is essential not only for data protection, but it also provides methods to apply aggregate invariant constraints to its components. This also means all invariant constraints must be applied to their associated *Add* and/or *Remove* methods.

Decision 10. Global types and constants exist as separate entities in the domain and design models. To implement global types and constants, however, they must eventually be associated with some class. This methodology encapsulates all global types and constants in a separate specification class.

Decision 11. All operations must return one and only one value. This means that multiple outputs must be consolidated into a single output class, and operations that have no real return value, such as procedures, return a boolean value. This restriction was necessary to reduce complexity in the functional transformation process.

Decision 12. The majority of aggregate transformations are inter-model, specifically from the specification to the design model. The focus on inter-model transformations is necessary because the objective of this research is to demonstrate the end-to-end transformation capabilities of **AFITTOOL**, and prior to this effort, no aggregate specification to design transforms had been defined.

5.4 Overall Aggregate Transformation Process

Overall, the methodology used to transform formal aggregate specifications to design had to perform the following basic operations:

1. Identify and transform the aggregate attributes to some representation in the design model.

2. Apply any aggregate invariant constraints to the appropriate attributes.

3. Transform the aggregate-specific operators to their corresponding methods,

4. Transform any aggregate-specific predicates to their corresponding program constructs.

Figure 21 outlines the sequence of transformations that occur on aggregate objects. The order and type (intra vs inter) of aggregate transformations are similar to the primitive operation transformations [16].

Transform 1 builds the initial aggregate attributes, transforms 2 through 8 apply invariant constraints to the aggregate attributes, transform 9 creates all supporting aggregate methods, transforms 10 through 13 replace aggregate operators with method calls, and transforms 14 and 15 transform aggregate predicates to an appropriate program construct. All aggregate transformations outlined in Figure 21 are explained in more detail in this chapter.

1. Domain aggregate attributes to design attributes.
2. Set inclusion of a component attribute invariant constraint.
3. Set non-inclusion of a component attribute invariant constraint.
4. Subset invariant constraint.
5. Disjoint set invariant constraint.
6. Cardinality less-than or less-than-equal invariant constraint.
7. Cardinality greater-than or greater-than-equal invariant constraint.
8. Declaration invariant constraint.
9. Set-to-set method build transformation.
10. Cardinality operator replacement.
11. Set inclusion/subset operator replacement.
12. Domain/range restriction operator replacement.
13. Set union, intersect, and difference operator replacement.
14. Universal predicate transformation.
15. Existential predicate transformation.

Figure 21 Aggregate Transformations

5.5 Container Specifications

Before any transformations occur, the containers that represent sets, sequences, and associations in the design model must be defined. This section outlines basic container requirements for all system containers used to represent sets, sequences, and associations. This section also defines the containers as implemented.

5.5.1 Container Requirements. The container classes, regardless of target language, must have minimum common characteristics, attributes, and functionality. First and foremost, all containers must apply the rules inherent to the aggregate attribute; that is, sets cannot have duplicates or an implied order, sequences do have an order and can have duplicates, and associations must meet the requirements outlined in Figure 25. Secondly, all containers must support a basic number of functions as defined in Figures 22, 23 and 24. The implementation specifics of these functions are left to the designer, which could eventually become a design refinement. This also does not preclude designers from extending the container class functionality.

5.5.2 Set Container Requirement Definition. The set container has the functions *GetFirst* and *GetNext*, which allow the caller to manually access the container's contents

(See Figure 22). *Add* adds an object to the set if the object is not already in the set, *Remove* removes a specified object from the set if the object is in the set, *Find* searches the structure for a particular object and returns the boolean value *true* if found and *false* otherwise, and *Size* returns the number of objects within the set.

Function Name	Parameters	Returns
GetFirst	Container	1st Object
GetNext	Container	Next Object
Add	Container; Object	Container with single object added*
Remove	Container; Object	Container with single object removed*
Find	Container; Object	Boolean
Size	Container	Natural

If the operation is successful

Figure 22 Mandatory Set Container Methods

5.5.3 Sequence Container Requirement Definition. The sequence container resembles the set container with the following differences: The function *Add* must append to the container list, and the addition of two functions: *GetLast*, which returns the last object in the list, and *Get*, which returns the object referenced by the subscript variable (See Figure 23).

Function Name	Parameters	Returns
GetFirst	Container	1st Object
GetNext	Container	Next Object
Getlast	Container	Last Object
Get	Container; Natural	Selected Object
Add	Container; Object	Container with object added
Remove	Container; Object	Container with object removed
Find	Container; Object	Boolean
Size	Container	Natural

Figure 23 Mandatory Sequence Container Methods

5.5.4 Association Container Requirement Definition. The domain and range of association containers share the same basic functions as the set container (See Figure 24). The important difference is that there are functions that must be overloaded. For instance, the *GetFirst* container can return the domain object, range object, or both,

depending on how it is referenced. Domain Restriction (*DomRes*) accepts the container class with a domain object value, and returns an association container with all members of the association with the same domain object. Range Restriction (*RanRes*) works the same way with the range.

Function Name	Parameters	Returns
GetFirst*	Container	1st Domain and Range Object
GetNext*	Container	Next Domain and Range Object
Add	Container; Domain Object; Range Object	Container with objects added
Remove	Container; Domain Object; Range Object	Container with objects removed
Find*	Container; Domain Object; Range Object	Boolean
Size	Container	Natural
DomRes	Container; Domain Object	Container
RanRes	Container; Range Object	Container

*Overloaded Operation

Figure 24 Mandatory Association Container Methods

Set and sequence container class structure and functionality are fairly straightforward to understand and implement. Associations build on sets and sequences, yet the number of association types and their corresponding attributes make this a more complex task to implement. Figure 25 lists the characteristics that each association container's domain and range must have with regard to its multiplicity, whether membership is mandatory, and if the domain or range should allow duplicates.

Symbol	Domain Mult	Domain Mandatory?	Domain Duplicates?	Range Mult	Range Mandatory?	Range Duplicates?
$\leftrightarrow$	Many	No	Yes	Many	No	Yes
$\rightarrow$	Many	No	No	One	Yes	Yes
$\nrightarrow$	Many	No	No	Optional	No	Yes
$\subset$	Optional	No	No	One	Yes	No
$\not\subset$	Optional	No	No	Optional	No	No
$\subset\!\!\rightarrow$	One	Yes	No	One	Yes	No
$\twoheadrightarrow$	Plus	Yes	No	One	Yes	Yes
$\nrightarrow\!\!\!\!\rightarrow$	Plus	Yes	No	Optional	No	Yes

Figure 25 Association Domain and Range Composition

From Figure 25, for example, we can tell that a total function (→) association's domain does not allow duplicates, that many members of the domain can map to the same range member, and that the range allows duplicates. Range membership is mandatory, domain is not. From this information, a key observation can be made. There are a number of relations that model single class relationships in the domain, range, or both. These types of relationships are correctly modeled as containers in this effort, yet a container implementation may not be the most efficient way to implement these relationships. Future design refinements could be used to explore the possibility of implementing one or two-way pointers in the domain and range classes.

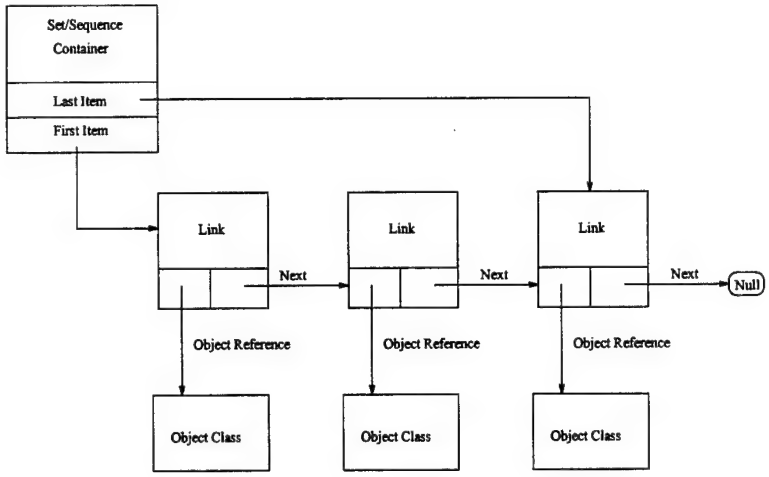


Figure 26 Canonical Set and Sequence Data Structure

5.5.5 Containers As Implemented. Overall, there is a container type for sets, sequences, and every type of association. All containers in this implementation provide the corresponding mandatory methods. For this effort, all mandatory container methods can be represented in the design model–that is, all program constructs used in the generic methods can be found in the design model. Aggregate containers for this effort were implemented using linked lists, as illustrated in Figure 26. The container attributes themselves do not reference the actual elements in the set, sequence, or association, but *link* classes (See Figure 27). Finally, containers in this effort were implemented using Ada generic packages. These generic packages were instantiated with a copy of each object's record and record access type. Link objects for sets and sequences consist of two attributes: a link

object, or a reference to the actual class within the list; and link next, an attribute that references the next link object. Link objects for associations add an additional reference for the range of the association. Link objects for associative object associations extend the associative object with an additional reference for the associative object. The implemented set container has two attributes, references to the first and last link object accessed within the list (see Figure 28).

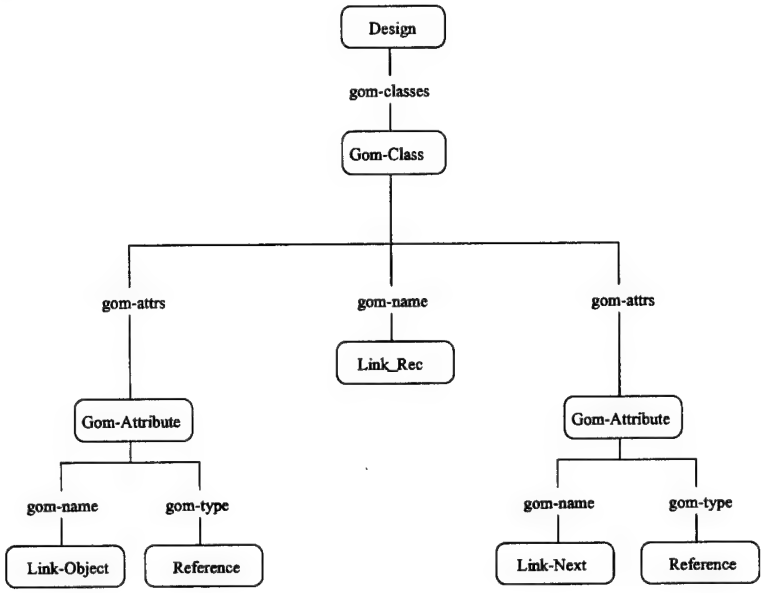


Figure 27 Link Instance Diagram

The sequence container also has the first and last attributes as mentioned previously, as well as a reference to the last link object in the list. Associative objects have the same structure as a set or sequence container, depending on which structure is being modeled.

5.6 Aggregate Component Attribute Transformations

With the containers defined, the transformations can now be defined. Each transformation identifies what is transformed, the rationale for the transformation, and any other pertinent information.

Aggregate Transform 1 - Transform Domain Aggregate Attributes to Design Attributes. The first transformation that occurs takes the aggregate component attributes from the specification domain to the design domain. These transformations

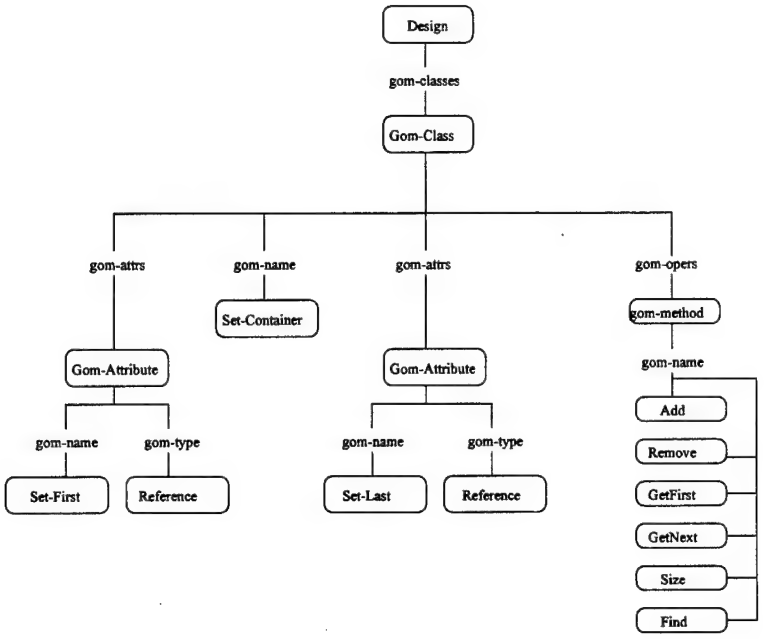


Figure 28 Canonical Set Instance Diagram

involve the four attributes that are unique to aggregate classes: Individual components, sets of components, sequences of components, and associations between component classes. Table 29 lists the specification attribute declarations and their corresponding design representation.

As discussed previously, aggregate components (with the exception of the single component attribute) are modeled using containers. Decision 8 from Section 2 of this chapter, *System Design Decisions*, states that all aggregate component attributes can only be added and removed by methods defined in the aggregate. *Add* and *Remove* for aggregate attributes have the following inherent characteristics: The cardinality of an aggregate attribute can only be zero or more (zero or one for a single component), and the *Add* and *Remove* methods are functionally equivalent to increment and decrement procedures, respectively.

Creating *Add* and *Remove* operations is straightforward. For a single component *Add*, its simply an assignment statement. Likewise, a single item *Remove* method resets the component attribute to a null value. *Add* and *Remove* methods for attributes implemented as containers use the *Add* and *Remove* methods supplied with the container class.

Symbol	Domain Model Representation(Mult)	Design Model Representation
Class	has-component(one)	Attribute Reference Type
P	has-component(many)	Generic Set Container
seq	has-component(many) and is-ordered	Generic Sequence Container
$\leftrightarrow$	has-association(many-many)	Generic Relation Container
$\nrightarrow$	has-association(many-optional)	Generic PartialFunction Container
$\rightarrow$	has-association(many-one)	Generic Function Container
$\rightarrowtail$	has-association(optional-one)	Generic Injection Container
$\rightarrowtail$	has-association(optional-optional)	Generic PartialInjection Container
$\twoheadrightarrow$	has-association(plus-one)	Generic Surjection Container
$\nrightarrow\!\!\!\!\twoheadrightarrow$	has-association(plus-optional)	Generic Partial Surjection Container
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	has-association(one-one)	Generic Bijection Container

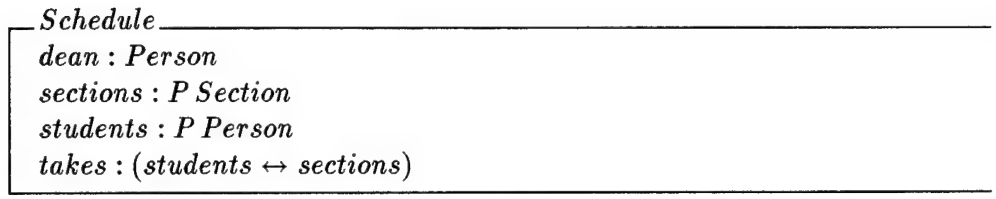
Figure 29 Specification to Design Model Mappings - Aggregate Attributes

Example of aggregate attribute transformation. Figure 30 is an example of how aggregate types are transformed from specification to code. Here, a *Z* schema called *Schedule* has the following declarations: the single component attribute *dean* of the class **Person**, the two sets *sections* and *students* of the classes **Section** and **Person**, respectively, and the relational association *takes*, with the domain over the set *students* and the range over the set *sections*.

The resulting design representation in Figure 30 begins with the inclusion of all generic container and component classes. The attribute *dean* has been defined as a reference type to the component class **Person**. The remaining attributes, *sections*, *students*, and *takes*, are declared as generic class types. Finally, all attributes have corresponding *Add* and *Remove* methods created within the domain.

5.7 Aggregate Invariant Constraint Transformation

The second round of transformations involve transforming aggregate invariant constraints. A number of observations can be made about aggregate invariant constraints. First, all invariant constraints handled can be reduced to a relationship. Aggregate attribute relationships are based either on relationships between objects, or on the cardinality of the attribute. Finally, all of the invariant constraints can potentially apply to both side of the relationship, as demonstrated in each example.

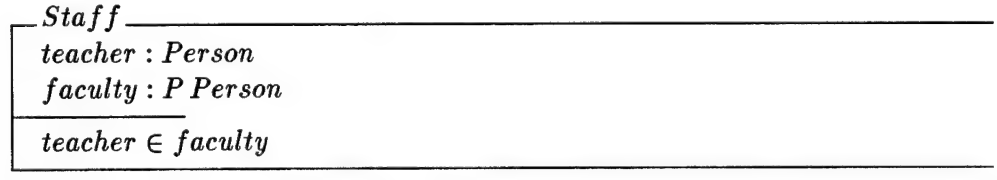



```
Design Model
Class Schedule Definition
    Has visibility to the generic classes Set_Container, Relation_Container
    Has visibility to the component classes Section, Person
    Has attribute dean        of type reference Person
    Has attribute sections    of type generic class Aggregate_Container
    Has attribute students    of type generic class Aggregate_Container
    Has attribute takes       of type generic class Relation_Container
    Has Methods Add_dean, Remove_dean, Add_sections, Remove_sections,
                Add_students, Remove_students, Add_takes, Remove_takes
End Class Schedule Definition
```

Figure 30 Aggregate Attribute Transformation

Invariant constraints over aggregate attributes must be transformed to pre-conditions within their applicable *Add* and *Remove* methods. The following definitions capture the primary aggregate invariant constraint transformations. All target methods for the invariant constraints are identified.

Aggregate Transform 2 - Set membership of a component attribute invariant constraint. Here, a single aggregate component must be a member of a set.



In this example, the invariant constraint *teacher* ∈ *faculty* would be added to the *Add_Teacher* method of *Staff*.

Add_Teacher

$\Delta Staff$
$input? : Person$

$input? \in faculty$
$teacher' = input?$

Also, the invariant constraint $input? \neq teacher$ must be added to the *Remove_Faculty* method of *Staff* to preserve the integrity of the original invariant constraint.

Remove_Faculty

$\Delta Staff$
$input? : Person$

$input? \neq teacher$
$faculty' = faculty \setminus \{input?\}$

In both methods, the lefthand side of the set inclusion operator was replaced with the input parameter name. In the *Remove_Faculty* method of *Staff*, the set inclusion operator was replaced with the $\neq$ operator.

Aggregate Transform 3 - Set non-inclusion of a component attribute invariant constraint. This invariant constraints states that the single component cannot be a member of a particular set.

Class

$teacher : Person$
$studentbody : P\, Person$

$teacher \notin studentbody$

Here, the invariant constraint $teacher \notin studentbody$ would be added to the *Add_Teacher* method of *Class*.

Add_Teacher

$\Delta Class$
$input? : Person$

$input? \notin studentbody$
$teacher' = input?$

An invariant constraint ensuring an object being added to the set *studentbody* is not the object *teacher* is added to the *Add_Studentbody* method of *Class*.

Add_Studentbody

$\Delta Class$
$input? : Person$

$input? \neq teacher$
$studentbody' = studentbody \cup \{input?\}$

Aggregate Transform 4 - Subset invariant constraint. This invariant constraint states that one set is the subset of another.

Company

$workforce : P\, Person$
$hourlyworkers : P\, Person$

$hourlyworkers \subset workforce$

In this example, the invariant constraint $hourlyworkers \subset workforce$ would be transformed to the *element of* invariant constraint in the *Add_Hourlyworkers* method of *Company*.

Add_Hourlyworkers

$\Delta Company$
$input? : Person$

$input? \in workforce$
$hourlyworkers' = hourlyworkers \cup \{input?\}$

The *not element of* invariant constraint is added to the *Remove_Workforce* method of *Company*.

Remove_Workforce

$\Delta Company$
$input? : Person$

$input? \notin hourlyworkers$
$workforce' = workforce \setminus \{input?\}$

Aggregate Transform 5 - Disjoint set invariant constraint. This invariant constraint does not allow two sets to have any shared members.

Company

$union : P\ Person$

$management : P\ Person$

$\forall x : management \bullet x \notin union$

In this example, the universal quantifier invariant constraint would be transformed into the *not element of* operator and added to the *Add_Union* method of *Company*.

Add_Union

$\Delta Company$

$input? : Person$

$input? \notin management$

$union' = union \cup \{input?\}$

It would also have to be added to the *Add_Management* method of *Company*.

Add_Management

$\Delta Company$

$input? : Person$

$input? \notin union$

$management' = management \cup \{input?\}$

Aggregate Transform 6 - Cardinality less-than or less-than-equal invariant constraint. Invariant constraints that have cardinality operators must take into consideration the fact that in an implementation, what's true in the pre-condition may be made false by the post-condition. For instance, there is an invariant constraint stating that the cardinality of **SetA** must be less than **SetB**. Now, the *Add* has been invoked with **SetA** having 39 members, and **SetB** having 40. This situation does not violate the invariant constraint, and the operation is executed. After the *Add* operation, however, the sets will be equal, violating the precondition. Therefore, all cardinality preconditions must check the the cardinality of a set plus one when adding, or the cardinality of a set minus one when subtracting. In this invariant constraint, the cardinality of a set can be expressed on either side of relational operators **less-than** or **less-than-equal**.

Auditorium

$seats : P\ seat$
$audience : P\ Person$

$\#audience \leq \#seats$

In this example, the invariant constraint would be added to the *Add_Audience* method of *Auditorium*.

Add_Audience

$\Delta Auditorium$
$input? : Person$

$\#audience + 1 \leq \#seats$
$audience' = audience \cup \{input?\}$

It would also have to be added to the *Remove_Seats* method of *Auditorium*.

Remove_Seats

$\Delta Auditorium$
$input? : Person$

$\#audience \leq \#seats - 1$
$seats' = seats \setminus \{input?\}$

Aggregate Transform 7 - Cardinality greater-than or greater-than-equal invariant constraint. This transformation is the inverse of transformation 6.

Auditorium

$seats : P\ seat$
$audience : P\ Person$

$\#seats \geq \#audience$

In this example, the same invariant constraints would be added as in transform 6.

Aggregate Transform 8 - Declaration invariant constraint. This particular transformation does not have an actual invariant constraint defined. Rather, the invariant constraint is captured within a declaration.

$$
\begin{array}{|l}
\hline
\textit{CourseOffering} \\
\textit{students} : \mathbb{P}\, \textit{Person} \\
\textit{classes} : \mathbb{P}\, \textit{Course} \\
\textit{takes} : (\textit{students} \leftrightarrow \textit{classes}) \\
\hline
\end{array}
$$

The *Z* schema *CourseOffering* illustrates this type of invariant constraint. In this example, the association *takes* has been written over the sets *students* and *classes*. Since an invariant constraint does not exist, equivalent invariant constraints must be constructed.

$$
\begin{array}{|l}
\hline
\textit{Add\_takes} \\
\Delta \textit{CourseOffering} \\
\textit{student?} : \textit{Person} \\
\textit{class?} : \textit{Course} \\
\hline
\textit{student?} \in \textit{students} \land \textit{class?} \in \textit{classes} \\
\textit{takes}' = \textit{takes} \cup \{(\textit{student?}, \textit{class?})\} \\
\hline
\end{array}
$$

Here, the *Z* functional schema *Add_takes* has been defined with the appropriate invariant constraints applied. Additionally, invariant constraints to enure non-membership in the association would also have to be added to the *Remove_Students* and *Remove_Classes* methods of *CourseOffering*.

$$
\begin{array}{|l}
\hline
\textit{Remove\_Students} \\
\Delta \textit{CourseOffering} \\
\textit{input?} : \textit{Person} \\
\hline
\textit{input?} \notin \operatorname{dom} \textit{takes} \\
\textit{students}' = \textit{students} \setminus \{\textit{input?}\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Remove\_Classes} \\
\Delta \textit{CourseOffering} \\
\textit{input?} : \textit{Course} \\
\hline
\textit{input?} \notin \operatorname{ran} \textit{takes} \\
\textit{classes}' = \textit{classes} \setminus \{\textit{input?}\} \\
\hline
\end{array}
$$

It is important to note there are a number of equivalent invariant constraints that are not mentioned in the above transforms. In the interest of simplicity, straightforward exam-

ples were used to illustrate the transformations. These transformations can also apply to sequences and the domain and ranges of associations, where applicable.

5.8 Set, Sequence, and Association Operations Transformation

The next step in the transformation process is to transform all aggregate operations into functions and their respective function calls. Figure 31 lists the aggregate attribute operations, how they're represented in the specification domain, and what they map to in the design model. The table lists a number of set, sequence and association operations that have been defined in the container classes. Container functions are used to implement aggregate operations whenever allowed because they are one-to-one or near one-to-one transforms. These functions can be implemented because the operations relate directly to a single entity. For example, if the specification asks for the cardinality of a set, the transformation is from the cardinality of a set to the *size* function of the set. Three specific instances for the operators **union**, **difference**, and **concatenation** also translate directly into container methods (refer to Figure 31, entries annotated "Single object").

Symbol	Specification Model Representation	Design Model Representation
#	Cardinality-expr	Container Function Size
$\in$	Relational1-Pred	Container Find
$\subset$	Relational1-Pred	Container Find
$\subseteq$	Relational1-Pred	Container Find
$\cup$	SetUnion-expr	Function Union Container Function Add(single object)
$\cap$	SetIntersect-expr	Function Intersection
$\setminus$	SetMinus-expr	Function Difference Container Function Remove(single object)
$\mathcal{C}$	dres	Container DomRes
$\mathcal{B}$	rres	Container RanRes
$\frown$	Concatenation	Function Concatenation Container Function Add(single object)
head	headseq	Container GetFirst
last	lastseq	Container Getlast

Figure 31 Specification Symbol to Design Model Mappings - Functional

Aggregate Transform 9 - Set-to-set method build transformation. Reviewing strategies to transform set, sequence, and association operators as functions, the easiest way would be to define all functions within the containers. Unfortunately, all set operators cannot be transformed this way. Operations where sets, sequences and associations are used to produce a set cannot be included within a container class because the input sets are frequently from distinct types. For example, a predicate may have been defined where an output set is the result of the union of some set and the range of an association. Since the set and association are different instantiations of container classes, the **union** operator cannot be contained in either generic class. The algorithms of these types of functions must therefore be generated.

```
function Union  (Container1 : in Container1_Type;
                 Container2 : in Container2_Type)
                 return  Container_Out_Type is

  Component     :  Component_Reference_Type :=  Null;
  Container_Out :  Container_Out_Type;
begin
  GetFirst (Container1, Component);
  while Component  /=  null
  loop
    Add (Container_Out, Component);
    GetNext (Container1, Component);
  end loop;
  GetFirst (Container2, Component);
  while Component  /=  null
  loop
    Add (Container_Out, Component);
    GetNext (Container2, Component);
  end loop;
  return Container_Out;
end Union;
```

Figure 32 Method Union Algorithm

Figures 32, 33, and 34 are the methods generated for the operations **union**, **intersect**, and **difference**, respectively. An Ada surface syntax has been used to express these algorithms for ease of understanding. Figure 32 represents the method *union*, which iteratively adds components from two input sets to an output set. The output set container structurally

restricts duplicate objects from being included. The standard container methods *GetFirst*, *Add*, and *GetNext* are used to manipulate the input and output containers.

```
function Intersect (Container1 : in Container1_Type;
                    Container2 : in Container2_Type)
                    return  Container_Out_Type is

  Component     :  Component_Reference_Type :=  Null;
  Container_Out :  Container_Out_Type;
begin
  GetFirst (Container1, Component);
  while Component  /=  null
  loop
    if Find ( Container2, Component)  =  true then
      Add (Container_Out, Component);
    end if;
    GetNext (Container1, Component);
  end loop;
  return Container_Out;
end Intersect;
```

Figure 33 Method Intersect Algorithm

The method *intersect* iterates through the first set and searches the second set using the container function *find*. If the object is found in both sets, the object is added using the *Add* method of the output set container. The method *difference* has the same behavior as intersect, except the object is added to the output set if the object is in the first set, but not in the second.

The algorithms are straightforward by design. This simplicity not only aids in the algorithm's understandability, but it also allows the algorithms to be represented canonically in the design model. These examples also do not display the complexity of deriving correct input and output parameters. It's important here to recall that if an input parameter is an association container, then any corresponding container calls within the function are overloaded to retrieve the correct domain or range object. (The exception is when the domain and range objects within the association are of the same type. Then the functions that retrieve both the domain and range are used, and the correct object is used in the function) The key transformation issue, then, is to ensure the correct attribute types are captured for use in the generated method.

```
function Difference  (Container1 : in Container1_Type;
                      Container2 : in Container2_Type)
                      return  Container_Out_Type is

  Component     :  Component_Reference_Type :=  Null;
  Container_Out :  Container_Out_Type;

begin
  GetFirst (Container1, Component);
  while Component  /=  null
  loop
    if Find (Container2, Component)  =  false then
       Add (Container_Out, Component);
    end if;
    GetNext (Container2, Component);
  end loop;
  return Container_Out;
end Difference;
```

Figure 34 Method Difference Algorithm

Now that all of the set operators have been represented as functions, the next set of transformations recursively find and replace all set operators within the method predicates with their respective function calls.

Aggregate Transform 10 - Cardinality operator replacement. In this transformation, all Cardinality operators are transformed to the container function call *Size* within the resulting container class.

Add_Audience

$\Delta Auditorium$
$input? : Person$

$\#audience + 1 \leq \#seats$
$audience' = audience \cup \{input?\}$

In this example, the cardinality operators are replaced by size method calls.

Add_Audience

$\Delta Auditorium$
$input? : Person$

$size(audience) + 1 \leq size(seats)$
$audience' = audience \cup \{input?\}$

Aggregate Transform 11 - Set membership/Subset operator replacement. In this transformation, all Set inclusion and Subset operators are transformed to the container function call *Find* within the resulting container class. If there is a domain and/or range in the relationship, the correct object type is determined from the association definition.

Add_Classes

$\Delta CourseOffering$
$input? : Course$

$input? \in \text{ran}\, takes$
$classes' = classes \cup \{input?\}$

In this example, the precondition states that the input object must exist in the range of the association *takes* before it can be added to the set *classes*. The invariant constraint is transformed into the following function call.

Add_Classes

$\Delta CourseOffering$
$input? : Course$

$find(takes, input?)$
$classes' = classes \cup \{input?\}$

In this transformation, the set operator *ran* is dropped, as the class type of the input resolves which of the overloaded *find* methods to call.

Aggregate Transform 12 - Domain/Range restriction operator replacement. In this transformation, all domain and range restriction operators are transformed to either the container function call *DomRes* or *RanRes* within the resulting container class.

Courses_Taken

$\Delta Schedule$
$input? : Student$
$classes! : P\, class$

$classes! = \{input?\}\, C\, takes$

The method *Courses_Taken* demonstrates how the domain of the association *takes* can be restricted with a single value, in this case *input?*. The resulting range set is returned in the output set *classes!* .

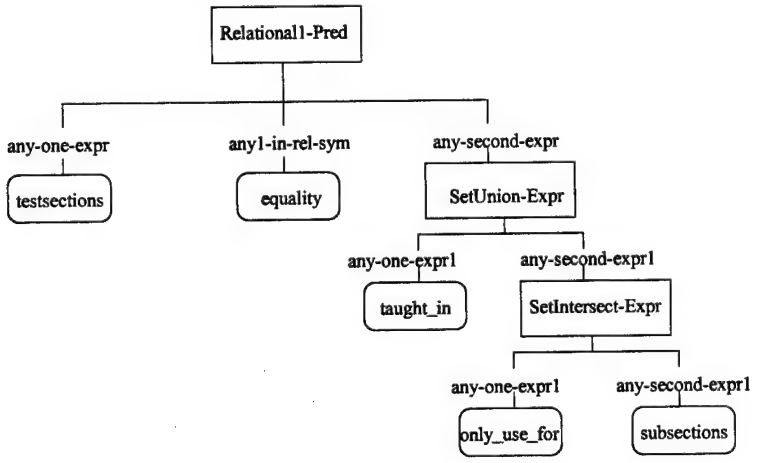
Courses_Taken

$\Delta Schedule$
$input? : Student$
$classes! : P\, class$

$classes! = DomRes(takes, input?)$

The domain restriction operator is replaced with the method call **DomRes** (provided by the assocation container) and the expressions on both sides of the original operator are added as parameters to the method call.

Aggregate Transform 13 - Set union, intersect, and difference operator replacement. Here, the **union**, **intersect**, and **difference** operators are replaced with function calls to the functions generated in transform 9. For example, refer to Figure 35.



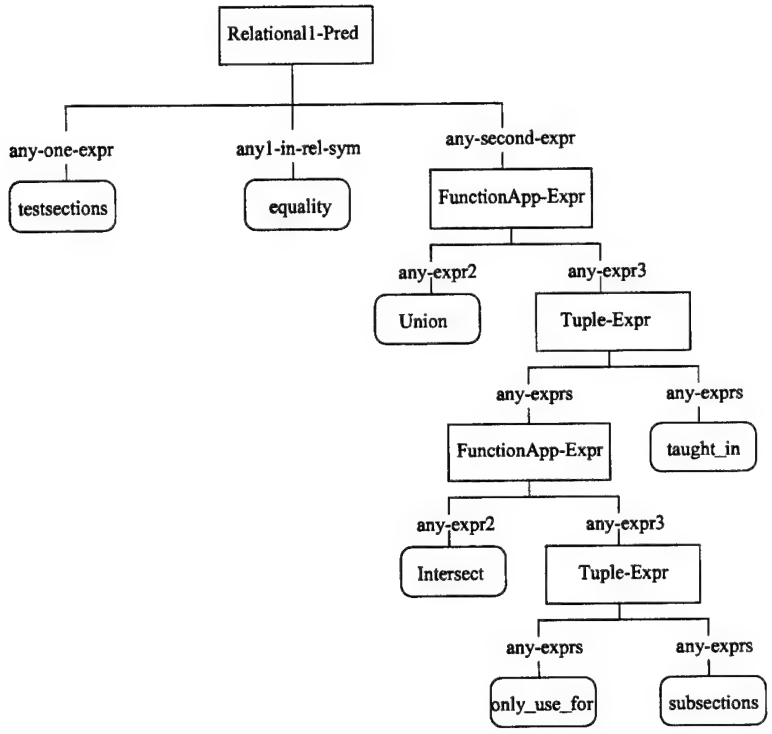
Get_Sections_Taken

$test\_sections! = taught\_in \cup only\_use\_for \cap subsections$

Figure 35 Aggregate Predicate Represented in the Domain AST

Figure 35 is a *Z* predicate with the set operators *Union* and *Intersect* in a post-condition. After transformation 9, a *Union* method would have been created with the

attributes *taught_in* and *only_use_for* as input parameters, and returns a set of type **Section**. Also, an *Intersect* method would have been created with a temporary input attribute set of **Section**, and the attribute *subsections*. The output is a set of type **Section**.



Get_Sections_Taken

$test\_sections! = Intersect(Union(taught\_in, only\_use\_for), subsections)$

Figure 36 Aggregate Predicate Represented in the Domain AST after Transform 13

Recall from Figure 35 how the post-condition containing set operators populated the domain model. Transform 13 redefines the post-condition to the representation shown in Figure 36.

5.9 Aggregate Predicate and Expression Transformation

The final transforms are performed in the design model, and in conjunction with the primitive transforms developed by Tankersley [16]. Since the aggregate attributes and their operators have been transformed to methods and method calls, no further transformations

at the expression level are required. There are, however, two predicates that only pertain to aggregate attributes: the **universal** and the **existential** predicates.

Aggregate Transform 14 - Universal Predicate Transformation. Universal predicates in this context are pre-conditions within methods that apply to all members of the entity expressed in the universal quantification. Therefore, this predicate closely maps to an iteration construct.

Give_Pay_Raise

$$\Delta Company$$

$$\forall x : workforce \bullet x'.pay = x.pay * 1.048$$

In the functional schema *Give_Pay_Raise* above, an universal quantifier has been applied to the set *workforces* to perform some function, in this case giving everyone in the workforce a payraise. This universal quantifier predicate transforms to the following design surface syntax:

```
current_worker = Get_First(workforce)
while current_worker not equal null
  Iterate
    current_worker.pay = current_worker.pay * 1.048
    current_worker = get_next(workforce)
```

Aggregate Transform 15 - Existential Predicate Transformation. An **existential** predicate maps closest to a search algorithm. Recall that each container has a *Find* method, which provides a search for an object reference within the container. An existential predicate, therefore, allows the specifier to elaborate search criteria.

Find_Available_Room

$$\Delta Schedule$$
$$selected\_room! : Room$$

$$\exists x : rooms \bullet x \notin scheduled \land selected\_room! = x$$

The method *Find_Available_Room* has an existential quantifier that states that if there exists an object in the set *room* that is not in the set *scheduled*, then return that object.

```
rooms.Get_First(room)
while not found or not end of list
  Iterate
    if find(scheduled, room) = true then
      selected_room = room
    end if
    rooms.get_next(room)
```

5.10 Summary

This chapter outlines a methodology to transform formal *Z* specifications to code. It details assumptions and design considerations used for the methodology. It defines how aggregate specification components map to their design counterparts, as well as their corresponding operations. The generic classes used to represent aggregate components have been defined. Finally, a number of transforms have been outlined to process aggregate specifications through the system.

VI. Conclusions and Possible Research

6.1 Results

This research effort of studying the feasibility of transforming formal aggregate specifications to a design model and, ultimately, to an implementation language, produced a number of positive results. The following paragraphs represent the major accomplishments resulting from this research. References to more detailed information are included in the heading, as applicable.

Aggregate object definitions were developed for the specification model (See Chapter 3). Sets, sequences, and associations are correctly represented in the domain model. Operators relating to these aggregate components are represented using predicates and expressions. Finally, an event mapping structure was defined to represent the dynamic aspects of aggregate objects.

Aggregate object definitions were developed for the design model (See Chapter 4). Class references were added to the model to represent aggregate component classes. Set, sequence, and association operators are represented as methods in the design model. Generic classes were incorporated into the model to define the canonical classes used to represent aggregate components.

Aggregate mappings between the specification and design models were identified (See Chapter 5). Single component class attributes in the specification are mapped to class reference attributes in the design. Set, sequence, and association attributes in the specification map to generic container classes. Aggregate operators are mapped to design methods. If the operator pertains to a single class, the method is defined within the generic container class. If the operator relates to two distinct aggregate component attributes, then the operator method is created and placed in the aggregate class.

Transformations for aggregate object invariant constraints were identified (See Chapter 5). Common aggregate invariant constraints are transformed, first to their appropriate specification methods, then from the specification to the design.

Transformations for aggregate operators were defined. Set, sequence, and association operators were transformed from the specification model to their design method counterparts.

Design model mappings to an Ada surface Syntax were identified. The appropriate Ada constructs for class, class references, and generic packages were mapped from the design model to the Ada specification and body grammars.

The end-to-end transformation of specifications was demonstrated (See Appendices). The automatic transformation of formal aggregate specifications was demonstrated for a subset of aggregate structural and functional attributes using a set of test specifications. The Ada code generated from these test specifications was then compiled and linked using the ObjectAda Version 7.1 compiler. Additionally, tests were written for the generic container classes to ensure proper operation. All test specifications and code modules are included in the final code turn-in.

Finally, a noteable observation can made with regard to the complete transformation of the aggregate specifications. The integrated software development approach provided by **AFITTOOL** reduces or eliminates inconsistencies between aggregate specification, design, and implementation representations because all specification elements can be directly traced all the way to their implementation counterparts, and all implementation elements produced by the specification have traceability to their originating specification.

6.2 Limitations

The successes of this research must be tempered with the known shortcomings of the effort. The large scope of the problem, combined with the relatively short amount of time allowed to address the problem domain were the limiting factors of the effort. Finally, these limitations were largely known up front, and have been identified throughout this document as assumptions or as follow-on research.

Transformation system 'brittleness'. The primary objective of this research was to demonstrate the complete end-to-end transformation of aggregate specifications. While transformations were demonstrated for a number of aggregate entities, these demonstra-

tions were by no means exhaustive. While assuming that all specifications are correct is essential for a proof of concept research effort such as this, it is not realistic for any system intended for operational use.

Other aggregate components and operations. This research focused on four primary aggregate components: single components, sets, sequences, and associations. There are other aggregate components and operations that were not explored in this research. For a complete listing of available formal aggregate components and operations see [3].

The dynamic model. Other than the event map representation defined in the specification, no dynamic aspects of the aggregate specifiaction are transformed.

6.3 Lessons Learned

A few observations can be made regarding this research that will help any future research efforts involving software specification development.

The components that comprise a complete transformation system such as **AFIT-TOOL** are numerous, diverse and complex. The domain knowledge of each component contained within a transformation system needed for efforts such as this one can easily overwhelm the researcher. Scoping the effort must therefore be of primary concern when performing this type of research.

There were a few assumptions made in this research effort that were incorrect. The first incorrect assumption involved the collection of a representative sample of aggregate specifications for testing. Finding correct aggregate predicates for testing proved to be exceedingly difficult. Parser difficulties, combined with incorrect *latex* to specification grammar mappings and the lack of work using aggregate objects, impeded the testing process.

Finally, assuming the transformation of aggregate components was largely independent of primitive objects was incorrect. Defining when and where aggregate transformations occur was often dependent on the corresponding primitive object transformations. Early in this research, for example, the decision was made to use *Add* and *Remove* methods to access class attributes. It was decided the use of these methods constituted a design

decision, and that these methods should therefore exist in the design model only. Primitive objects also use methods to access their attributes, yet the decision was made to transform these methods in the specification. Had these two transformations been truly independent, both approaches would have been valid. Unfortunately, non-set operations exist within aggregate-related predicates, and, therefore, must have primitive transformations applied. To illustrate this point, refer to figure 37.

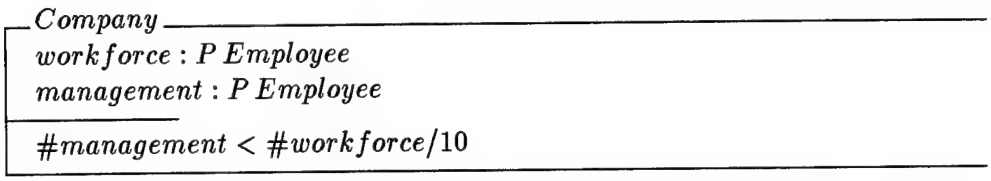


Figure 37 Aggregate and Primitive Operation Example

In this example, an invariant constraint has been defined that states the size of *management* must be less than the size of *workforce* divided by ten. With the original approach, the cardinality operators would have been transformed directly to the design tree, and primitive operation transforms would either been ignored or would have to be duplicated in the aggregate transforms.

6.4 Follow-on Research

As with any research effort, a number of related research areas are either uncovered during the effort, or topics originally identified for research must be scoped out due to time constraints. The following paragraphs identify a number of areas this study revealed as possible follow-on research.

Aggregate specification refinement. With the exception of Anderson's Elicitor-Harvestor work, few specification refinement transforms exist [1]. Examples of specification model refinement include but are not limited to performing specification-specific inconsistency correction, such as detecting and correcting subsumed relationships, and specification refinements to the structural, dynamic, and functional aspects of the specification model.

Design refinements. There is a great opportunity to research possible design refinements. This research could explore design refinements on all aspects of the design model, from container optimization to migrating class references from containers to primitive objects, algorithm refinements, and class structure optimizations.

I/O Interfaces. Noe conducted extensive research involving the integration of **CASE** tools with **AFITTOOL** [11]. Research opportunities still exist exploring ways to interface the design model with standardized, readily available software components. Representing I/O bindings of external software entities such as databases and graphical user interfaces as classes in the design model would greatly extend the tool's design refinement capabilities.

Storage of system evolution. The current system operates dynamically; that is, the only persistent data stores of the system exist in the input *Z* specifications and the output Ada surface syntax. The feasibility of storing system information at various points in the transformation process could be studied. This research could lead to the ability to perform such operations as "undoing" transformations, replaying the transformation process, and storing useful transformations for future use.

Dynamic model transformation. As stated in Chapter 5, dynamic model transformations were not addressed in this research. Research could therefore be performed to demonstrate the transformation of dynamic specifications to design.

Extending aggregate specification to design transforms. This research focused on transforming a workable subset of abstract aggregate entities. Follow-on research could extend the aggregate transformation set to include less common set representations and operators, such as bags, anti-domain restrictions, and set-former notation.

6.5 Summary

This research effort provides a solid foundation for representing and transforming aggregate specifications to code, fully supporting the feasibility of such a transformation system.

Appendix A. Attribute Transformation Example - Specification
Aggregate Attribute Transformation

Object Name: Aggregate_Xform1

Object Number: 980308

Object Description: This is a specification that demonstrates all the possible aggregate attributes and their transformation.

Date: 02/26/99

History: Date: 08/04/98 (Kissack) initial creation

Author: Kissack

Superclass: None

Components: See below

Context: None

Attributes: None

Constraints: None

Z Static Schema:

Aggregate_Xform1

$teacher : Faculty$
$sections : P\ Section$
$subsections : P\ Section$
$students : P\ Student$
$alsostudents : P\ Student$
$rooms : P\ Room$
$faculty : P\ Faculty$
$timeslots : P\ Timeslot$
$takes : (students \leftrightarrow sections)$
$only\_use\_for : (sections \rightarrow rooms)$
$teaches : (sections \nrightarrow faculty)$
$has\_office : (faculty \twoheadrightarrow rooms)$
$taught\_in : (Room \nrightarrow sections)$
$must\_be\_taught\_in : (rooms \Subset sections)$
$taught\_at : (timeslots \subset sections)$
$must\_be\_taught\_at : (timeslots \subseteq sections)$

$true$

```
--Written by jkissack on 3/3/1999
With udtypes, Ada.Strings.Unbounded;
  Use udtypes, Ada.Strings.Unbounded;
  With  Faculty, Section, Student, Room, Timeslot;
Use  Faculty, Section, Student, Room, Timeslot;
With
  UNCHECKED_DEALLOCATION, AGGREGATE_CONTAINER,
  RELATION_CONTAINER, FUNCTION_CONTAINER,
  PARTIALFUNCTION_CONTAINER, SURJECTION_CONTAINER,
  PARTIALSURJECTION_CONTAINER, BIJECTION_CONTAINER,
  INJECTION_CONTAINER, PARTIALINJECTION_CONTAINER;
package AGGREGATE_XFORM1 is
  package NEW_SECTION  is new  AGGREGATE_CONTAINER
    ( SECTION_REC, SECTION_REFERENCE ); use  NEW_SECTION;
  package NEW_STUDENT  is new  AGGREGATE_CONTAINER
    ( STUDENT_REC, STUDENT_REFERENCE ); use  NEW_STUDENT;
  package NEW_ROOM  is new  AGGREGATE_CONTAINER
    ( ROOM_REC, ROOM_REFERENCE ); use  NEW_ROOM;
  package NEW_TIMESLOT  is new  AGGREGATE_CONTAINER
    ( TIMESLOT_REC, TIMESLOT_REFERENCE ); use  NEW_TIMESLOT;
  package NEW_TAKES  is new  RELATION_CONTAINER
    ( STUDENT_REC, SECTION_REC, STUDENT_REFERENCE,
      SECTION_REFERENCE
    ); use  NEW_TAKES;
  package NEW_ONLY_USE_FOR  is new  FUNCTION_CONTAINER
    ( SECTION_REC, ROOM_REC, SECTION_REFERENCE, ROOM_REFERENCE
    ); use  NEW_ONLY_USE_FOR;
  package NEW_TEACHES  is new  PARTIALFUNCTION_CONTAINER
    ( SECTION_REC, FACULTY_REC, SECTION_REFERENCE,
      FACULTY_REFERENCE
    ); use  NEW_TEACHES;
  package NEW_HAS_OFFICE  is new  SURJECTION_CONTAINER
    ( FACULTY_REC, ROOM_REC, FACULTY_REFERENCE, ROOM_REFERENCE
    ); use  NEW_HAS_OFFICE;
  package NEW_TAUGHT_IN  is new  PARTIALSURJECTION_CONTAINER
    ( ROOM_REC, SECTION_REC, ROOM_REFERENCE, SECTION_REFERENCE
    ); use  NEW_TAUGHT_IN;
  package NEW_MUST_BE_TAUGHT_IN  is new  BIJECTION_CONTAINER
    ( ROOM_REC, SECTION_REC, ROOM_REFERENCE, SECTION_REFERENCE
    ); use  NEW_MUST_BE_TAUGHT_IN;
  package NEW_TAUGHT_AT  is new  INJECTION_CONTAINER
    ( TIMESLOT_REC, SECTION_REC, TIMESLOT_REFERENCE,
      SECTION_REFERENCE
    ); use  NEW_TAUGHT_AT;
  package NEW_MUST_BE_TAUGHT_AT  is new
```

```
    PARTIALINJECTION_CONTAINER
    ( TIMESLOT_REC, SECTION_REC, TIMESLOT_REFERENCE,
      SECTION_REFERENCE
    ); use  NEW_MUST_BE_TAUGHT_AT;
Type AGGREGATE_XFORM1_REC  is tagged private;
  Type AGGREGATE_XFORM1_REFERENCE is access AGGREGATE_XFORM1
  _REC;
--methods
  procedure ADD_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure REMOVE_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure ADD_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure REMOVE_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure ADD_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    );
  procedure REMOVE_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    );
  procedure ADD_ALSOSTUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    );
  procedure REMOVE_ALSOSTUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    );
  procedure ADD_ROOMS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in ROOM_REFERENCE
    );
  procedure REMOVE_ROOMS
```

```
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in ROOM_REFERENCE
  );
procedure ADD_FACULTY
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in FACULTY_REFERENCE
  );
procedure REMOVE_FACULTY
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in FACULTY_REFERENCE
  );
procedure ADD_TIMESLOTS
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in TIMESLOT_REFERENCE
  );
procedure REMOVE_TIMESLOTS
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in TIMESLOT_REFERENCE
  );
procedure ADD_TAKES
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in STUDENT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure REMOVE_TAKES
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in STUDENT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure ADD_ONLY_USE_FOR
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  );
procedure REMOVE_ONLY_USE_FOR
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  );
procedure ADD_TEACHES
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in FACULTY_REFERENCE
  );
procedure REMOVE_TEACHES
```

```
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in FACULTY_REFERENCE
  );
procedure ADD_HAS_OFFICE
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in FACULTY_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  );
procedure REMOVE_HAS_OFFICE
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in FACULTY_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  );
procedure ADD_TAUGHT_IN
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure REMOVE_TAUGHT_IN
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure ADD_MUST_BE_TAUGHT_IN
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure REMOVE_MUST_BE_TAUGHT_IN
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure ADD_TAUGHT_AT
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in TIMESLOT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure REMOVE_TAUGHT_AT
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in TIMESLOT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  );
procedure ADD_MUST_BE_TAUGHT_AT
```

```
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_DOMAIN : in TIMESLOT_REFERENCE;
      LINK_RANGE : in SECTION_REFERENCE
    );
  procedure REMOVE_MUST_BE_TAUGHT_AT
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_DOMAIN : in TIMESLOT_REFERENCE;
      LINK_RANGE : in SECTION_REFERENCE
    );
  procedure ADD_TEACHER
    ( Z_AGGREGATE_XFORM1 : in out AGGREGATE_XFORM1_REC;
      Z_TEACHER : in Faculty;
      AGGREGATE_XFORM1 : out AGGREGATECLASS
    );
  procedure REMOVE_TEACHER
    ( Z_AGGREGATE_XFORM1 : in out AGGREGATE_XFORM1_REC;
      AGGREGATE_XFORM1 : out AGGREGATECLASS
    );
  Private  Type  AGGREGATE_XFORM1_REC  is TAGGED RECORD
  TEACHER :  FACULTY_REFERENCE;
  SECTIONS :  NEW_SECTION.AGGREGATE_CONTAINER_REC;
  SUBSECTIONS :  NEW_SECTION.AGGREGATE_CONTAINER_REC;
  STUDENTS :  NEW_STUDENT.AGGREGATE_CONTAINER_REC;
  ALSOSTUDENTS :  NEW_STUDENT.AGGREGATE_CONTAINER_REC;
  ROOMS :  NEW_ROOM.AGGREGATE_CONTAINER_REC;
  FACULTY :  NEW_FACULTY.AGGREGATE_CONTAINER_REC;
  TIMESLOTS :  NEW_TIMESLOT.AGGREGATE_CONTAINER_REC;
  TAKES :  NEW_TAKES.RELATION_CONTAINER_REC;
  ONLY_USE_FOR :  NEW_ONLY_USE_FOR.FUNCTION_CONTAINER_REC;
  TEACHES :  NEW_TEACHES.PARTIALFUNCTION_CONTAINER_REC;
  HAS_OFFICE :  NEW_HAS_OFFICE.SURJECTION_CONTAINER_REC;
  TAUGHT_IN :  NEW_TAUGHT_IN.PARTIALSURJECTION_CONTAINER_REC;
  MUST_BE_TAUGHT_IN :
    NEW_MUST_BE_TAUGHT_IN.BIJECTION_CONTAINER_REC;
  TAUGHT_AT :  NEW_TAUGHT_AT.INJECTION_CONTAINER_REC;
  MUST_BE_TAUGHT_AT :
    NEW_MUST_BE_TAUGHT_AT.PARTIALINJECTION_CONTAINER_REC;
END RECORD; end AGGREGATE_XFORM1;
```

```
--Written by jkissack on 3/3/1999
with Ada.Text_IO, Ada.Strings.Unbounded;
  use Ada.Text_IO, Ada.Strings.Unbounded; package body
  AGGREGATE_XFORM1 is --local constants --methods
  procedure ADD_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    NEW_SECTION.add ( CONTAINER.SECTIONS, LINK_OBJECT);
  end ADD_SECTIONS;
  procedure REMOVE_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    NEW_SECTION.remove ( CONTAINER.SECTIONS, LINK_OBJECT);
  end REMOVE_SECTIONS;
  procedure ADD_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    NEW_SECTION.add ( CONTAINER.SUBSECTIONS, LINK_OBJECT);
  end ADD_SUBSECTIONS;
  procedure REMOVE_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    NEW_SECTION.remove
      ( CONTAINER.SUBSECTIONS, LINK_OBJECT);
  end REMOVE_SUBSECTIONS;
  procedure ADD_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    ) is begin
    NEW_STUDENT.add ( CONTAINER.STUDENTS, LINK_OBJECT);
  end ADD_STUDENTS;
  procedure REMOVE_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    ) is begin
    NEW_STUDENT.remove ( CONTAINER.STUDENTS, LINK_OBJECT);
  end REMOVE_STUDENTS;
  procedure ADD_ALSOSTUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM1_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    ) is begin
```

```
  NEW_STUDENT.add ( CONTAINER.ALSOSTUDENTS, LINK_OBJECT);
end ADD_ALSOSTUDENTS;
procedure REMOVE_ALSOSTUDENTS
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in STUDENT_REFERENCE
  ) is begin
  NEW_STUDENT.remove
    ( CONTAINER.ALSOSTUDENTS, LINK_OBJECT);
end REMOVE_ALSOSTUDENTS;
procedure ADD_ROOMS
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in ROOM_REFERENCE
  ) is begin
  NEW_ROOM.add ( CONTAINER.ROOMS, LINK_OBJECT);
end ADD_ROOMS;
procedure REMOVE_ROOMS
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in ROOM_REFERENCE
  ) is begin
  NEW_ROOM.remove ( CONTAINER.ROOMS, LINK_OBJECT);
end REMOVE_ROOMS;
procedure ADD_FACULTY
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in FACULTY_REFERENCE
  ) is begin
  NEW_FACULTY.add ( CONTAINER.FACULTY, LINK_OBJECT);
end ADD_FACULTY;
procedure REMOVE_FACULTY
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in FACULTY_REFERENCE
  ) is begin
  NEW_FACULTY.remove ( CONTAINER.FACULTY, LINK_OBJECT);
end REMOVE_FACULTY;
procedure ADD_TIMESLOTS
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in TIMESLOT_REFERENCE
  ) is begin
  NEW_TIMESLOT.add ( CONTAINER.TIMESLOTS, LINK_OBJECT);
end ADD_TIMESLOTS;
procedure REMOVE_TIMESLOTS
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_OBJECT : in TIMESLOT_REFERENCE
  ) is begin
  NEW_TIMESLOT.remove
    ( CONTAINER.TIMESLOTS, LINK_OBJECT);
```

```
end REMOVE_TIMESLOTS;
procedure ADD_TAKES
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in STUDENT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_TAKES.add
    ( CONTAINER.TAKES, LINK_DOMAIN, LINK_RANGE);
end ADD_TAKES;
procedure REMOVE_TAKES
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in STUDENT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_TAKES.remove
    ( CONTAINER.TAKES, LINK_DOMAIN, LINK_RANGE);
end REMOVE_TAKES;
procedure ADD_ONLY_USE_FOR
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  ) is begin
  NEW_ONLY_USE_FOR.add
    ( CONTAINER.ONLY_USE_FOR, LINK_DOMAIN, LINK_RANGE);
end ADD_ONLY_USE_FOR;
procedure REMOVE_ONLY_USE_FOR
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  ) is begin
  NEW_ONLY_USE_FOR.remove
    ( CONTAINER.ONLY_USE_FOR, LINK_DOMAIN, LINK_RANGE);
end REMOVE_ONLY_USE_FOR;
procedure ADD_TEACHES
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in FACULTY_REFERENCE
  ) is begin
  NEW_TEACHES.add
    ( CONTAINER.TEACHES, LINK_DOMAIN, LINK_RANGE);
end ADD_TEACHES;
procedure REMOVE_TEACHES
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in SECTION_REFERENCE;
    LINK_RANGE : in FACULTY_REFERENCE
```

```
  ) is begin
  NEW_TEACHES.remove
    ( CONTAINER.TEACHES, LINK_DOMAIN, LINK_RANGE);
end REMOVE_TEACHES;
procedure ADD_HAS_OFFICE
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in FACULTY_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  ) is begin
  NEW_HAS_OFFICE.add
    ( CONTAINER.HAS_OFFICE, LINK_DOMAIN, LINK_RANGE);
end ADD_HAS_OFFICE;
procedure REMOVE_HAS_OFFICE
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in FACULTY_REFERENCE;
    LINK_RANGE : in ROOM_REFERENCE
  ) is begin
  NEW_HAS_OFFICE.remove
    ( CONTAINER.HAS_OFFICE, LINK_DOMAIN, LINK_RANGE);
end REMOVE_HAS_OFFICE;
procedure ADD_TAUGHT_IN
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_TAUGHT_IN.add
    ( CONTAINER.TAUGHT_IN, LINK_DOMAIN, LINK_RANGE);
end ADD_TAUGHT_IN;
procedure REMOVE_TAUGHT_IN
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_TAUGHT_IN.remove
    ( CONTAINER.TAUGHT_IN, LINK_DOMAIN, LINK_RANGE);
end REMOVE_TAUGHT_IN;
procedure ADD_MUST_BE_TAUGHT_IN
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_MUST_BE_TAUGHT_IN.add
    ( CONTAINER.MUST_BE_TAUGHT_IN, LINK_DOMAIN, LINK_RANGE);
end ADD_MUST_BE_TAUGHT_IN;
procedure REMOVE_MUST_BE_TAUGHT_IN
```

```
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in ROOM_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_MUST_BE_TAUGHT_IN.remove
    ( CONTAINER.MUST_BE_TAUGHT_IN, LINK_DOMAIN, LINK_RANGE);
end REMOVE_MUST_BE_TAUGHT_IN;
procedure ADD_TAUGHT_AT
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in TIMESLOT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_TAUGHT_AT.add
    ( CONTAINER.TAUGHT_AT, LINK_DOMAIN, LINK_RANGE);
end ADD_TAUGHT_AT;
procedure REMOVE_TAUGHT_AT
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in TIMESLOT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_TAUGHT_AT.remove
    ( CONTAINER.TAUGHT_AT, LINK_DOMAIN, LINK_RANGE);
end REMOVE_TAUGHT_AT;
procedure ADD_MUST_BE_TAUGHT_AT
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in TIMESLOT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_MUST_BE_TAUGHT_AT.add
    ( CONTAINER.MUST_BE_TAUGHT_AT, LINK_DOMAIN, LINK_RANGE);
end ADD_MUST_BE_TAUGHT_AT;
procedure REMOVE_MUST_BE_TAUGHT_AT
  ( CONTAINER : in out AGGREGATE_XFORM1_REC;
    LINK_DOMAIN : in TIMESLOT_REFERENCE;
    LINK_RANGE : in SECTION_REFERENCE
  ) is begin
  NEW_MUST_BE_TAUGHT_AT.remove
    ( CONTAINER.MUST_BE_TAUGHT_AT, LINK_DOMAIN, LINK_RANGE);
end REMOVE_MUST_BE_TAUGHT_AT;
procedure ADD_TEACHER
  ( CONTAINER : in out SCHEDULE_REC;
    Z_TEACHER : in Faculty_Reference) is begin
        CONTAINER.TEACHER := Z_TEACHER;
end ADD_TEACHER;
procedure REMOVE_TEACHER
```

```
    ( CONTAINER : in out SCHEDULE_REC;
      Z_TEACHER : in Faculty_Reference) is begin
     CONTAINER.TEACHER := Null;
  end REMOVE_TEACHER;
end AGGREGATE_XFORM1;
```

Appendix B. Transforms 4, 5, 6, and 7 Examples - Specification
Aggregate Transformation Examples

Object Name: Aggregate_Xform4

Object Number: 980308

Object Description: This is a specification that demonstrates aggregate transforms 4, subset invariant constraint transform, 5, disjoint set invariant constraint transform, and 6 and 7, cardinality invariant constraints.

Date: 02/26/99

History: Date: 08/04/98 (Kissack) initial creation

Author: Kissack

Superclass: None

Components: See below

Context: None

Attributes: None

Constraints: None

Z Static Schema:

Aggregate_Xform4

$$
\begin{array}{|l}
sections : P\ Section \\
subsections : P\ Section \\
students : P\ Student \\
takes : (students \leftrightarrow sections) \\
\hline
\forall x : subsections \bullet x \in sections \\
\#sections > 5 \\
\#subsections < \#sections \\
\forall x : (\mathrm{ran}\ takes) \bullet x \notin subsections \\
\hline
\end{array}
$$

```
--Written by jkissack on 3/3/1999
With udtypes, Ada.Strings.Unbounded;
  Use udtypes, Ada.Strings.Unbounded;
  With  Section, Student;
Use  Section, Student;
With
  UNCHECKED_DEALLOCATION, AGGREGATE_CONTAINER,
  RELATION_CONTAINER;
package AGGREGATE_XFORM4 is
  package NEW_SECTION  is new  AGGREGATE_CONTAINER
    ( SECTION_REC, SECTION_REFERENCE ); use  NEW_SECTION;
  package NEW_STUDENT  is new  AGGREGATE_CONTAINER
    ( STUDENT_REC, STUDENT_REFERENCE ); use  NEW_STUDENT;
  package NEW_TAKES  is new  RELATION_CONTAINER
    ( STUDENT_REC, SECTION_REC, STUDENT_REFERENCE,
      SECTION_REFERENCE
    ); use  NEW_TAKES;
Type AGGREGATE_XFORM4_REC  is tagged private;
  Type AGGREGATE_XFORM4_REFERENCE is access AGGREGATE_XFORM4
  _REC;
--methods
  procedure ADD_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure REMOVE_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure ADD_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure REMOVE_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    );
  procedure ADD_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    );
  procedure REMOVE_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    );
```

```
  procedure ADD_TAKES
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_DOMAIN : in STUDENT_REFERENCE;
      LINK_RANGE : in SECTION_REFERENCE
    );
  procedure REMOVE_TAKES
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_DOMAIN : in STUDENT_REFERENCE;
      LINK_RANGE : in SECTION_REFERENCE
    );
  Private Type AGGREGATE_XFORM4_REC is TAGGED RECORD
  SECTIONS : NEW_SECTION.AGGREGATE_CONTAINER_REC;
  SUBSECTIONS : NEW_SECTION.AGGREGATE_CONTAINER_REC;
  STUDENTS : NEW_STUDENT.AGGREGATE_CONTAINER_REC;
  TAKES : NEW_TAKES.RELATION_CONTAINER_REC;
END RECORD; end AGGREGATE_XFORM4;
```

```
--Written by jkissack on 3/3/1999
with Ada.Text_IO, Ada.Strings.Unbounded;
  use Ada.Text_IO, Ada.Strings.Unbounded; package body
  AGGREGATE_XFORM4 is --local constants --methods
  procedure ADD_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    NEW_SECTION.add ( CONTAINER.SECTIONS, LINK_OBJECT);
  end ADD_SECTIONS;
  procedure REMOVE_SECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    if ((SIZE ( CONTAINER.SUBSECTIONS)  <
           SIZE ( CONTAINER.SECTIONS)  -  1
         )  and  ( SIZE ( CONTAINER.SECTIONS)  -  1  >  5)
        )  and  ( FIND ( SUBSECTIONS, LINK_OBJECT)  /=  true)
   then NEW_SECTION.remove
          ( CONTAINER.SECTIONS, LINK_OBJECT);
     else Ada.Text_IO.Put_Line( "Remove failed");
    end if;
  end REMOVE_SECTIONS;
  procedure ADD_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    if ((FINDRAN ( TAKES, LINK_OBJECT)  /=  true )  and
          ( SIZE ( CONTAINER.SUBSECTIONS)  +  1  <
              SIZE ( CONTAINER.SECTIONS))
        )  and  ( FIND ( SECTIONS, LINK_OBJECT)  =  true)
   then NEW_SECTION.add
          ( CONTAINER.SUBSECTIONS, LINK_OBJECT);
     else Ada.Text_IO.Put_Line( "Add failed");
    end if;
  end ADD_SUBSECTIONS;
  procedure REMOVE_SUBSECTIONS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in SECTION_REFERENCE
    ) is begin
    NEW_SECTION.remove
      ( CONTAINER.SUBSECTIONS, LINK_OBJECT);
  end REMOVE_SUBSECTIONS;
  procedure ADD_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
```

```
      LINK_OBJECT : in STUDENT_REFERENCE
    ) is begin
    NEW_STUDENT.add ( CONTAINER.STUDENTS, LINK_OBJECT);
  end ADD_STUDENTS;
  procedure REMOVE_STUDENTS
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_OBJECT : in STUDENT_REFERENCE
    ) is begin
    NEW_STUDENT.remove ( CONTAINER.STUDENTS, LINK_OBJECT);
  end REMOVE_STUDENTS;
  procedure ADD_TAKES
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_DOMAIN : in STUDENT_REFERENCE;
      LINK_RANGE : in SECTION_REFERENCE
    ) is begin
    if FIND ( SUBSECTIONS, LINK_RANGE)  /=  true
   then NEW_TAKES.add
          ( CONTAINER.TAKES, LINK_DOMAIN, LINK_RANGE);
     else Ada.Text_IO.Put_Line( "Add failed");
    end if;
  end ADD_TAKES;
  procedure REMOVE_TAKES
    ( CONTAINER : in out AGGREGATE_XFORM4_REC;
      LINK_DOMAIN : in STUDENT_REFERENCE;
      LINK_RANGE : in SECTION_REFERENCE
    ) is begin
    NEW_TAKES.remove
      ( CONTAINER.TAKES, LINK_DOMAIN, LINK_RANGE);
  end REMOVE_TAKES;
end AGGREGATE_XFORM4;
```

Appendix C. Transforms 9 and 13 Examples - Specification

Aggregate Transformation Examples

Object Name: Aggregate_Xform13

Object Number: 980308

Object Description: This is a specification that demonstrates aggregate transform 9, the creation of in-line aggregate operator methods, and transform 13, aggregate operator replacement.

Date: 02/26/99

History: Date: 08/04/98 (Kissack) initial creation

Author: Kissack

Superclass: None

Components: See below.

Context: None

Attributes: None

Constraints: None

Z Static Schema:

$$
\begin{array}{|l}
\hline
\textit{Aggregate\_Xform13} \\
\textit{management} : \mathbb{P}\, \textit{Person} \\
\textit{laborers} : \mathbb{P}\, \textit{Person} \\
\hline
\textit{true} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Get\_All\_Workers} \\
\Delta \textit{Aggregate\_Xform13} \\
\textit{workforce}! : \mathbb{P}\, \textit{Person} \\
\hline
\textit{workforce}! = \textit{management} \cup \textit{laborers} \\
\hline
\end{array}
$$

Get_Line_Mgrs

$\Delta Aggregate\_Xform13$

$line\_mgrs! : P\ Person$

$line\_mgrs! = management \cap laborers$

Get_Upper_Management

$\Delta Aggregate\_Xform13$

$upper\_management! : P\ Person$

$upper\_management! = management \setminus laborers$

```
--Written by jkissack on 3/6/1999
With udtypes, Ada.Strings.Unbounded;
  Use udtypes, Ada.Strings.Unbounded; With  Person;
Use  Person;
With  UNCHECKED_DEALLOCATION, AGGREGATE_CONTAINER;
package AGGREGATE_XFORM13 is
  package NEW_PERSON  is new  AGGREGATE_CONTAINER
    ( PERSON_REC, PERSON_REFERENCE ); use  NEW_PERSON;
Type AGGREGATE_XFORM13_REC  is tagged private;
  Type AGGREGATE_XFORM13_REFERENCE is access
  AGGREGATE_XFORM13_REC;
--methods
  function DIFFERENCE_PERSON_AND_PERSON
    ( CONTAINER1 : in NEW_Person.Aggregate_Container_Rec;
      CONTAINER2 : in NEW_Person.Aggregate_Container_Rec
    ) return  New_Person.Aggregate_Container_Rec;
  function INTERSECT_PERSON_AND_PERSON
    ( CONTAINER1 : in NEW_Person.Aggregate_Container_Rec;
      CONTAINER2 : in NEW_Person.Aggregate_Container_Rec
    ) return  New_Person.Aggregate_Container_Rec;
  function UNION_PERSON_AND_PERSON
    ( CONTAINER1 : in NEW_Person.Aggregate_Container_Rec;
      CONTAINER2 : in NEW_Person.Aggregate_Container_Rec
    ) return  New_Person.Aggregate_Container_Rec;
  procedure ADD_MANAGEMENT
    ( CONTAINER : in out AGGREGATE_XFORM13_REC;
      LINK_OBJECT : in PERSON_REFERENCE
    );
  procedure REMOVE_MANAGEMENT
    ( CONTAINER : in out AGGREGATE_XFORM13_REC;
      LINK_OBJECT : in PERSON_REFERENCE
    );
  procedure ADD_LABORERS
    ( CONTAINER : in out AGGREGATE_XFORM13_REC;
      LINK_OBJECT : in PERSON_REFERENCE
    );
  procedure REMOVE_LABORERS
    ( CONTAINER : in out AGGREGATE_XFORM13_REC;
      LINK_OBJECT : in PERSON_REFERENCE
    );
  function GET_ALL_WORKERS
    ( Z_AGGREGATE_XFORM13 : in AGGREGATE_XFORM13_REC ) return
    NEW_Person.Aggregate_Container_Rec;
  function GET_LINE_MGRS
    ( Z_AGGREGATE_XFORM13 : in AGGREGATE_XFORM13_REC ) return
```

```
    NEW_Person.Aggregate_Container_Rec;
  function GET_UPPER_MANAGEMENT
    ( Z_AGGREGATE_XFORM13 : in AGGREGATE_XFORM13_REC ) return
    NEW_Person.Aggregate_Container_Rec;
Private  Type  AGGREGATE_XFORM13_REC  is TAGGED RECORD
  MANAGEMENT :  NEW_PERSON.AGGREGATE_CONTAINER_REC;
  LABORERS :  NEW_PERSON.AGGREGATE_CONTAINER_REC;
END RECORD; end AGGREGATE_XFORM13;
```

```
--Written by jkissack on 3/6/1999
with Ada.Text_IO, Ada.Strings.Unbounded;
  use Ada.Text_IO, Ada.Strings.Unbounded; package body
  AGGREGATE_XFORM13 is --local constants --methods

  function DIFFERENCE_PERSON_AND_PERSON
    ( CONTAINER1 : in NEW_Person.Aggregate_Container_Rec;
      CONTAINER2 : in NEW_Person.Aggregate_Container_Rec
    ) return  New_Person.Aggregate_Container_Rec is
    TEMP_LINK_OBJECT :  Person_Reference :=  NULL;
    TEMP_CONTAINER :  NEW_Person.Aggregate_Container_Rec;
    TEMP_CONTAINER2 :  NEW_Person.Aggregate_Container_Rec;
    CONTAINER_OUT :  New_Person.Aggregate_Container_Rec;
  begin
    TEMP_CONTAINER := CONTAINER1;
    TEMP_CONTAINER2 := CONTAINER2;
    GETFIRST ( TEMP_CONTAINER, TEMP_LINK_OBJECT);
    while TEMP_LINK_OBJECT  /=  null loop
      if FIND ( TEMP_CONTAINER2, TEMP_LINK_OBJECT)  =  false
     then ADDITEM ( CONTAINER_OUT, TEMP_LINK_OBJECT);
      end if;
      GETNEXT ( TEMP_CONTAINER, TEMP_LINK_OBJECT);
    end loop;
    return CONTAINER_OUT;
  end DIFFERENCE_PERSON_AND_PERSON;

  function INTERSECT_PERSON_AND_PERSON
    ( CONTAINER1 : in NEW_Person.Aggregate_Container_Rec;
      CONTAINER2 : in NEW_Person.Aggregate_Container_Rec
    ) return  New_Person.Aggregate_Container_Rec is
    TEMP_LINK_OBJECT :  Person_Reference :=  NULL;
    TEMP_CONTAINER :  NEW_Person.Aggregate_Container_Rec;
    TEMP_CONTAINER2 :  NEW_Person.Aggregate_Container_Rec;
    CONTAINER_OUT :  New_Person.Aggregate_Container_Rec;
  begin
    TEMP_CONTAINER := CONTAINER1;
    TEMP_CONTAINER2 := CONTAINER2;
    GETFIRST ( TEMP_CONTAINER, TEMP_LINK_OBJECT);
    while TEMP_LINK_OBJECT  /=  null loop
      if FIND ( TEMP_CONTAINER2, TEMP_LINK_OBJECT)  =  true
     then ADDITEM ( CONTAINER_OUT, TEMP_LINK_OBJECT);
      end if;
      GETNEXT ( TEMP_CONTAINER, TEMP_LINK_OBJECT);
    end loop;
    return CONTAINER_OUT;
```

```
end INTERSECT_PERSON_AND_PERSON;

function UNION_PERSON_AND_PERSON
  ( CONTAINER1 : in NEW_Person.Aggregate_Container_Rec;
    CONTAINER2 : in NEW_Person.Aggregate_Container_Rec
  ) return  New_Person.Aggregate_Container_Rec is
  TEMP_LINK_OBJECT :  Person_Reference :=  NULL;
  TEMP_CONTAINER :  NEW_Person.Aggregate_Container_Rec;
  TEMP_CONTAINER2 :  NEW_Person.Aggregate_Container_Rec;
  CONTAINER_OUT :  New_Person.Aggregate_Container_Rec;
begin
  TEMP_CONTAINER := CONTAINER1;
  TEMP_CONTAINER2 := CONTAINER2;
  GETFIRST ( TEMP_CONTAINER, TEMP_LINK_OBJECT);
  while TEMP_LINK_OBJECT  /=  null loop
    ADDITEM ( CONTAINER_OUT, TEMP_LINK_OBJECT);
    GETNEXT ( TEMP_CONTAINER, TEMP_LINK_OBJECT);
  end loop;
  GETFIRST ( TEMP_CONTAINER2, TEMP_LINK_OBJECT);
  while TEMP_LINK_OBJECT  /=  null loop
    ADDITEM ( CONTAINER_OUT, TEMP_LINK_OBJECT);
    GETNEXT ( TEMP_CONTAINER2, TEMP_LINK_OBJECT);
  end loop;
  return CONTAINER_OUT;
end UNION_PERSON_AND_PERSON;

procedure ADD_MANAGEMENT
  ( CONTAINER : in out AGGREGATE_XFORM13_REC;
    LINK_OBJECT : in PERSON_REFERENCE
  ) is begin
  NEW_PERSON.add ( CONTAINER.MANAGEMENT, LINK_OBJECT);
end ADD_MANAGEMENT;

procedure REMOVE_MANAGEMENT
  ( CONTAINER : in out AGGREGATE_XFORM13_REC;
    LINK_OBJECT : in PERSON_REFERENCE
  ) is begin
  NEW_PERSON.remove ( CONTAINER.MANAGEMENT, LINK_OBJECT);
end REMOVE_MANAGEMENT;

procedure ADD_LABORERS
  ( CONTAINER : in out AGGREGATE_XFORM13_REC;
    LINK_OBJECT : in PERSON_REFERENCE
  ) is begin
  NEW_PERSON.add ( CONTAINER.LABORERS, LINK_OBJECT);
```

```
  end ADD_LABORERS;

  procedure REMOVE_LABORERS
    ( CONTAINER : in out AGGREGATE_XFORM13_REC;
      LINK_OBJECT : in PERSON_REFERENCE
    ) is begin
    NEW_PERSON.remove ( CONTAINER.LABORERS, LINK_OBJECT);
  end REMOVE_LABORERS;

  function GET_ALL_WORKERS
    ( Z_AGGREGATE_XFORM13 : in AGGREGATE_XFORM13_REC ) return
    NEW_Person.Aggregate_Container_Rec is begin
    return
      UNION_PERSON_AND_PERSON(Z_AGGREGATE_XFORM13.MANAGEMENT,
      Z_AGGREGATE_XFORM13.LABORERS);
  end GET_ALL_WORKERS;

  function GET_LINE_MGRS
    ( Z_AGGREGATE_XFORM13 : in AGGREGATE_XFORM13_REC ) return
    NEW_Person.Aggregate_Container_Rec is begin
    return
      INTERSECT_PERSON_AND_PERSON(Z_AGGREGATE_XFORM13.MANAGEMENT,
      Z_AGGREGATE_XFORM13.LABORERS);
  end GET_LINE_MGRS;

  function GET_UPPER_MANAGEMENT
    ( Z_AGGREGATE_XFORM13 : in AGGREGATE_XFORM13_REC ) return
    NEW_Person.Aggregate_Container_Rec is begin
    return
      DIFFERENCE_PERSON_AND_PERSON(Z_AGGREGATE_XFORM13.MANAGEMENT,
      Z_AGGREGATE_XFORM13.LABORERS);
  end GET_UPPER_MANAGEMENT;
end AGGREGATE_XFORM13;
```

Bibliography

[1] Anderson, Gary L. *An Interactive Tool for Refining Software Specifications from a Formal Domain Model.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Mar 1999. AFIT/GCS/ENG/99M-01.

[2] Beem, Charles G. *Transforming Algebraically-Based Object Models Into a Canonical Form for Design Refinement.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Dec 1995. AFIT/GCS/ENG/95D-01, AD-A303 748.

[3] Ben Potter, Jane Sinclair and David Till. *Object Orientation in Z.* Englewood Cliffs, New Jersey: Prentice-Hall International, 1991.

[4] Blaha, Michael and William Premerlani. "A Catalog of Object Model Transformations." *Presented at 3rd Working Conference on Reverse Engineering.* November 1996.

[5] Coad and Yourdon. *Object-Oriented Analysis.* Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1991.

[6] Coad and Yourdon. *Object-Oriented Design.* Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1991.

[7] Cribbs, John, et al. *An Evaluation of Object-Oriented Analysis and Design Methodologies.* Raleigh, NC: SIGS Books, 1992.

[8] DeLoach, Scott A. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specification.* PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, June 1996. AFIT/DS/ENG/96-05, AD-A310 608.

[9] Hartrum, Thomas C. and Paul D. Bailor. "Teaching Formal Extensions of Informal-Based Object-Oriented Analysis Methodologies." *Proceedings Software Engineering Education (7th SEI CSEE Conference.* 389–409. Jan 1994.

[10] Lin, Catherine J. *Unification of Larch and Z-Based Object Models to Support Algebraically-Based Design Refinement: The Larch Perspective.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Dec 1994. AFIT/GCS/ENG/94D-15, AD-A289 235.

[11] Noe, Penelope. *A Structured Approach to Tool Integration.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Mar 1999. AFIT/GCS/ENG/99M-14.

[12] Rumbaugh, James, et al. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1991.

[13] Stepney, Susan, et al. *An Introduction to Formal Specification and Z.* Springer-Verlay, 1992.

[14] Sward, Ricky E. *Extracting Functionally Equivalent Object-Oriented Designs from Imperative Legacy Code*. PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, Sep 1997. AFIT/DS/ENG/97-04.

[15] Swartout, William and Robert Balzer. "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, *25*:41–45 (1982).

[16] Tankersley, Travis W. *Generating Executable Code from Formal Specifications of Primitive Objects*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Mar 1999. AFIT/GCS/ENG/99M-19.

[17] Wabiszewski, Kathleen M. *Unification of Larch and Z-Based Object Models to Support Algebraically-Based Design Refinement: The Z Perspective*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Dec 1994. AFIT/GCS/ENG/94D-24, AD-A289 234.

Vita

John Ashley Kissack was born May 28th, 1965, in Vancouver, Washington. He graduated from Anacortes High School, Anacortes, Washington, in 1983. He married Karen Olk, also from Anacortes, the same year. John enlisted in the Air Force the fall of 1983. He has had numerous assignments, including Maxwell AFB, Alabama, Lajes Field, The Azores, Portugal, and Kelly AFB, Texas. John received his undergraduate degree in Computer Science from Park College, Parkville, Missouri, in 1992. A distinguished graduate from Officer Training School, John was commissioned in 1995. John and Karen have four children: Ashley, Adam, Kyle, and Rebecca.

Permanent address: 804 Holbrook Drive
Newport News, Va 23602